1. MSP430 Overview
    1. Introduction
    2. MSP430 Main characteristics
    3. Address space
    4. Central Processing Unit (MSP430 CPU)
    5. Central Processing Unit (MSP430X CPU)
    6. Addressing modes
    7. MSP430 instruction set
2. Code Composer Essentials
    1. Code Composer Essentials
    2. Introduction to CCE IDE
    3. Creating a Project
    4. Code Editor
    5. File history
    6. Import and Export functionality
    7. Project Configuration details
    8. Introduction to Debug with CCE
3. General purpose Input/Output
    1. Laboratory GPIO: Lab1 - Blinking the LED
    2. Laboratory GPIO: Lab2 - Blinking the LED half the speed
    3. Laboratory GPIO: Lab3 - Toggle the LED state by pressing the push button
    4. Laboratory GPIO: Lab4 - Enable/disable LED blinking by push button press
4. Timers
    1. Laboratory Timers: Lab1 - Memory clock with Basic Timer1
    2. Laboratory Timers: Lab2 - Real Time Clock with Basic Timer1
    3. Laboratory Timers: Lab3 - Memory Clock with Timer_A
    4. Laboratory Timers: Lab4 - Buzzer tone generator

    5. Laboratory Timers: Lab5 - Frequency measurement
5. LCD Controller
    1. Laboratory LCD controller: Lab1 - LCD message display
6. Data Acquisition
    1. Laboratory Signal Acquisition: Lab1 - SAR ADC10 conversion
    2. Laboratory Signal Acquisition: Lab2 - SAR ADC12 conversion
    3. Laboratory Signal Acquisition: Lab3 - SD16_A ADC conversion
    4. Laboratory Signal Acquisition: Lab4 - Voltage signal comparison with Comparator_A
7. Digital-to-Analog Converter (DAC)
    1. Laboratory DAC: Lab1 - Voltage ramp generator
8. Direct Memory Access (DMA)
    1. Laboratory DMA: Lab1 - Data Memory transfer triggered by software
    2. Laboratory DMA: Lab2 - Sinusoidal waveform generator
9. Hardware Multiplier
    1. Laboratory Hardware Multiplier: Lab1 - Multiplication without hardware multiplier
    2. Laboratory Hardware Multiplier: Lab2 - Multiplication with hardware multiplier
    3. Laboratory Hardware Multiplier: Lab3 - RMS and active power calculation
10. Flash Programming
    1. Laboratory Flash memory: Lab1 - Flash memory programming with the CPU executing the code from flash memory
    2. Laboratory Flash memory: Lab2 - Flash memory programming with the CPU executing the code in RAM
11. Communication

1. Laboratory Communications: Lab1 - Echo test using the UART mode of the USCI module
2. Laboratory Communications: Lab2 - Echo test using SPI
3. Laboratory Communications: Lab3 - Echo test using I2C

Introduction

Introduction

The types of devices such as microprocessor, microcontroller, processor, digital signal processor (DSP), amongst others, in a certain manner, are related to the same device – the ASIC (Application Specific Integrated Circuit). Each processing device executes instructions, following a determined program applied to the inputs and shares architectural characteristics developed from the first microprocessors created in 1971. In the three decades after the development of the first microprocessor, huge developments and innovations have been made in this engineering field. Any of the terms used at the beginning of this section are correct to define a microprocessor, although each one has different characteristics and applications.

The definition of a microcontroller is somewhat difficult due to the constantly changing nature of the silicon industry. What we today consider a microcontroller with medium capabilities is several orders of magnitude more powerful, than the computer used on the first space missions. Nevertheless, some generalizations can be made as to what characterizes a microcontroller. Typically, microcontrollers are selected for embedded systems projects, i.e., control systems with a limited number of inputs and outputs where the controller is *embedded* into the system.

The programmable SoC (system-on-chip) concept started in 1972 with the 4-bit TMS1000 microcomputer developed by Texas Instruments (TI), and in those days it was ideal for applications such as calculators and ovens. This term was changed to Microcontroller Unit (MCU), which was more descriptive of a typical application. Nowadays, MCUs are at the heart of many physical systems, with higher levels of integration and processing power at lower power consumption.

The following list presents several qualities that define a microcontroller:

- Cost: Usually, the microcontrollers are high-volume, low cost devices;

- Clock frequency: Compared with other devices (microprocessors and DSPs), microcontrollers use a low clock frequency. Microcontrollers today can run up to 100 MHz/ 100 Million Instructions Per Second (MIPS)

- Power consumption: orders of magnitude lower than their DSP and MPU cousins;

- Bits: 4 bits (older devices) to 32 bits devices;

- Memory: Limited available memory, usually less than 1 MByte;

- Input/Output (I/O): Low to high (8-150) pin-out count.

Request the MSP430 Teaching ROM Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

MSP430 Main characteristics

MSP430 Main characteristics

Although there are variants in devices in the family, a MSP430 microcontroller can be characterized by:

- Low power consumption:

- 0.1 µA for RAM data retention;
- 0.8 µA for real time clock mode operation;
- 250 µA/MIPS at active operation.

Low operation voltage (from 1.8 V to 3.6 V).

< 1 µs clock start-up.

< 50 nA port leakage.

Zero-power Brown-Out Reset (BOR).

On-chip analogue devices:

- 10/12/16-bit Analogue-to-Digital Converter (ADC);
- 12-bit dual Digital-to-Analogue Converter (DAC);
- Comparator-gated timers;
- Operational Amplifiers (OP Amps);
- Supply Voltage Supervisor (SVS).

16 bit RISC CPU:

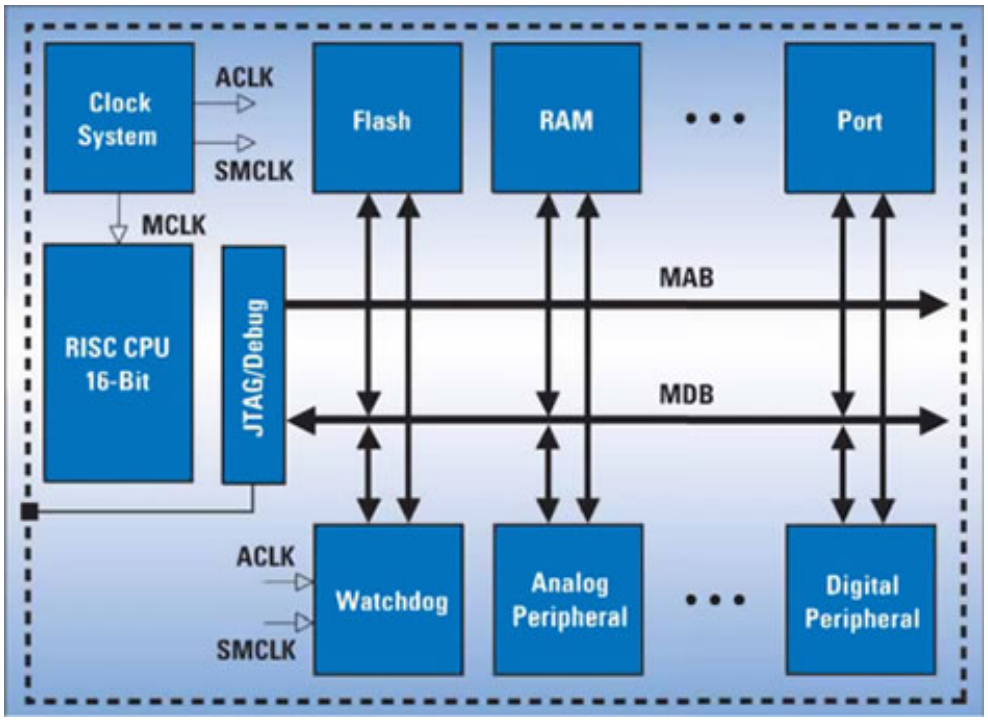
- Instructions processing on either bits, bytes or words;
- Compact core design reduces power consumption and cost;
- Compiler efficient;
- 27 core instructions;
- 7 addressing modes;
- Extensive vectored-interrupt capability.

Flexibility:

- Up to 256 kB In-System Programmable (ISP) Flash;
- Up to 100 pin options;
- USART, I2C, Timers;
- LCD driver;
- Embedded emulation.

The microcontroller's performance is directly related to the 16-bit data bus, the 7 addressing modes and the reduced instructions set, which allows a shorter, denser programming code for fast execution. These microcontroller families share a 16-bit CPU (Central Processing Unit) core, RISC type, intelligent peripherals, and flexible clock system that interconnects using a Von Neumanncommon memory address bus (MAB) and memory data bus (MDB) architecture.

MSP430 architecture.



Request the MSP430 Teaching ROM Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Address space

Address space

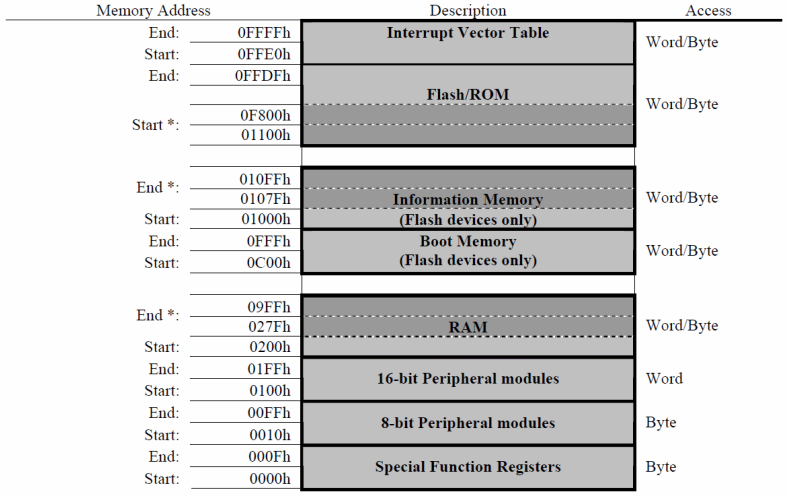
All memory, including RAM, Flash/ROM, information memory, special function registers (SFRs), and peripheral registers are mapped into a single, contiguous address space.

Note: See the device-specific datasheets for specific memory maps. Code access is always performed on even addresses. Data can be accessed as bytes or words.

The MSP430 is available with either Flash or ROM memory types. The memory type is identified by the letter immediately following “MSP430” in the part numbers.

Flash devices: Identified by the letter “F” in the part numbers, having the advantage that the code space can be erased and reprogrammed.

ROM devices: Identified by the letter “C” in the part numbers. They have the advantage of being very inexpensive because they are shipped pre-programmed, which is the best solution for high-volume designs.



* Depending on device family.

For all devices, each memory location is formed by 1 data byte. The CPU is capable of addressing data values either as bytes (8 bits) or words (16 bits). Words are always addressed at an even address, which contain the least significant byte, followed by the next odd address, which contains the most significant byte. For 8-bit operations, the data can be accessed from either odd or even addresses, but for 16-bit operations, the data values can only be accessed from even addresses.

Interrupt vector table

The interrupt vector table is mapped at the very end of memory space (upper 16 words of Flash/ROM), in locations 0FFE0h through to 0FFFEh (see the device-specific datasheets). The priority of the interrupt vector increases with the word address.

Interrupt vector table for MSP430 families.

Vector Address	Priority	**’ 11xx and ’ 12xx**	**’ 13x and ’ 14x**	‘2xx	**’ 3xx**	**’ 4xx**
0xFFFE	31, Highest	Hard Reset/ Watchdog	Hard Reset/ Watchdog	Hard Reset/ Watchdog	Hard Reset/ Watchdog	Hard R Watch
0xFFFC	30	Oscillator/ Flash/NMI	Oscillator/ Flash/NMI	Oscillator/ Flash/NMI	Oscillator/ Flash/NMI	Oscilla Flash/N
0xFFFA	29	Unused	Timer_B	Timer_B (22x2, 22x4, 23x, 24x, 26x only)	Dedicated I/O	Timer_ and’44
0xFFF8	28	Unused	Timer_B	Timer_B (22x2, 22x4, 23x, 24x only)	Dedicated I/O	Timer_ and’44
0xFFF6	27	Comparator	Comparator	Comparator_A+ (20x1 only, 21x1, 23x, 24x, 26x)	Unused	Compa
0xFFF4	26	Watchdog Timer	Watchdog Timer	Watchdog Timer+	Watchdog Timer	Watch Timer
0xFFF2	25	Timer_A	USART Rx	Timer_A	Timer_A	USAR Rx(’43 and’44
0xFFF0	24	Timer_A	USART0 Tx	Timer_A	Timer_A	USAR Tx(’43 and’44
0xFFEE	23	USART0 Rx (’12xx only)	ADC	USCI Rx(22x2, 22x4, 23x, 24x, 26x only)I2C status (23x, 24x)	USART Rx	ADC(’ and’44
0xFFEC	22	USART0 Tx (’12xx only)	Timer_A	USCI Tx(22x2, 22x4, 23x, 24x, 26x only)I2C Rx/Tx (23x, 24x, 26x only)	USART Tx	Timer_
0xFFEA	21	ADC10	Timer_A	ADC10 (20x2 22x2, 22x4 only)ADC12 (23x, 24x, 26x only)SD16_A (20x3 only)	ADC(’32x and ’33x) Timer/Port (’31x)	Timer_
0xFFE8	20	Unused	Port 1	USI(20x2, 20x3 only)	Timer/Port(’32x and ’33x)	Port 1
0xFFE6	19	Port 2	USART1	Port P2	Port 2	USAR

			Rx			Rx('44
0xFFE4	18	Port 1	USART1 Tx	Port P1	Port 1	USAR Tx('44
0xFFE2	17	Unused	Port 2	USCI Rx (23x, 24x, 26x only)I2C status (241x, 261x only)	Basic Timer	Port 2
0xFFE0	16	Unused	Unused	USCI Tx (23x,24x only)I2C Rx/Tx (241x, 261x only)	Port 0	Basic T
	15			DMA (241x, 261x only)		
	14			DAC12 (241x, 261 only)		
	13 to 0Lowest			Reserved		

Flash/ROM

The start address of Flash/ROM depends on the amount of Flash/ROM present on the device. The start address varies between 01100h (60k devices) to 0F800h (2k devices) and always runs to the end of the address space at location 0FFFFh. Flash can be used for both code and data. Word or byte tables can also be stored and read by the program from Flash/ROM.

All code, tables, and hard-coded constants reside in this memory space.

Information memory (Flash devices only)

The MSP430 flash devices contain an address space for information memory. It is like an onboard EEPROM, where variables needed for the next power up can be stored during power down. It can also be used as code memory. Flash memory may be written one byte or word at a time, but must be erased in segments. The information memory is divided into two 128-byte segments. The first of these segments is located at addresses 01000h through to 0107Fh (Segment B), and the second is at address 01080h through to 010FFh (Segment A). This is the case in 4xx devices. It is 256 bytes (4 segments of 64 bytes each) in 2xx devices.

Boot memory (Flash devices only)

The MSP430 flash devices contain an address space for boot memory, located between addresses 0C00h through to 0FFFh. The "bootstrap loader" is located in this memory space, which is an external interface that can be used to program the flash memory in addition to the JTAG. This memory region is not accessible by other applications, so it cannot be overwritten accidentally. The bootstrap loader performs some of the same functions as the JTAG interface (excepting the security fuse programming), using the TI data structure protocol for UART communication at a fixed data rate of 9600 baud.

RAM

RAM always starts at address 0200h. The end address of RAM depends on the amount of RAM present on the device. RAM is used for both code and data.

Peripheral Modules

Peripheral modules consist of all on-chip peripheral registers that are mapped into the address space. These modules can be accessed with byte or word instructions, depending if the peripheral module is 8-bit or 16-bit respectively. The 16-bit peripheral modules are located in the address space from addresses 0100 through to 01FFh and the 8-bit peripheral modules are mapped into memory from addresses 0010h through to 00FFh.

Special Function Registers (SFRs)

Some peripheral functions are mapped into memory with special dedicated functions. The Special Function Registers (SFRs) are located at memory addresses from 0000h to 000Fh, and are the specific registers for:

- Interrupt enables (locations 0000h and 0001h);

- Interrupt flags (locations 0002h and 0003h);

- Enable flags (locations 0004h and 0005h).

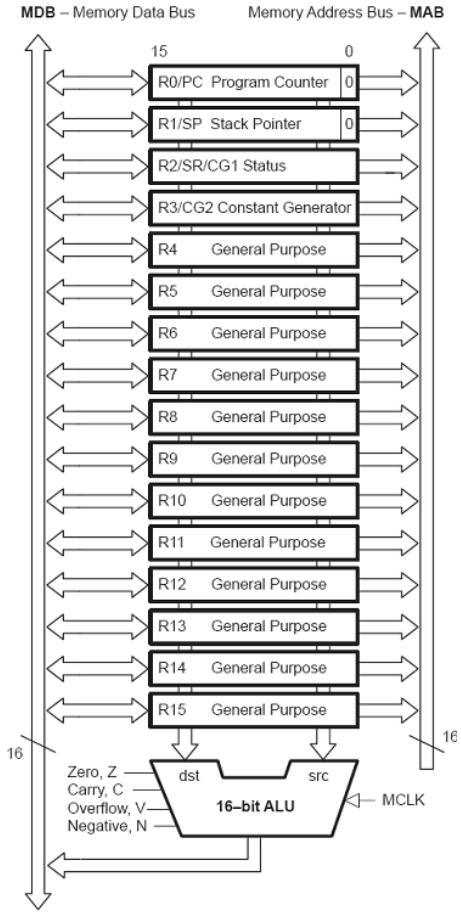
SFRs must be accessed using byte instructions only. See the device-specific data sheets for the applicable SFR bits.

Request the MSP430 Teaching ROM Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Central Processing Unit (MSP430 CPU)

The RISC type architecture of the CPU is based on a short instruction set (27 instructions), interconnected by a 3-stage instruction pipeline for instruction decoding. The CPU has a 16-bit ALU, four dedicated registers and twelve working registers, which makes the MSP430 a high performance microcontroller suitable for low power applications. The addition of twelve working general purpose registers saves CPU cycles by allowing the storage of frequently used values and variables instead of using RAM.

The orthogonal instruction set allows the use of any addressing mode for any instruction, which makes programming clear and consistent, with few exceptions, increasing the compiler efficiency for high-level languages such as C.
MSP430 CPU block diagram.



Arithmetic Logic Unit (ALU)

The MSP430 CPU includes an arithmetic logic unit (ALU) that handles addition, subtraction, comparison and logical (AND, OR, XOR) operations. ALU operations can affect the overflow, zero, negative, and carry flags in the status register.

MSP430 CPU registers

The CPU incorporates sixteen 16-bit registers:

- Four registers (R0, R1, R2 and R3) have dedicated functions;

- There are 12 working registers (R4 to R15) for general use.

R0: Program Counter (PC)

The 16-bit Program Counter (PC/R0) points to the next instruction to be read from memory and executed by the CPU. The Program counter is implemented by the number of bytes used by the instruction (2, 4, or 6 bytes, always even). It is important to remember that the PC is aligned at even addresses, because the instructions are 16 bits, even though the individual memory addresses contain 8-bit values.

R1: Stack Pointer (SP)

The Stack Pointer (SP/R1) is located in R1.

1st: stack can be used by user to store data for later use (instructions: store by PUSH, retrieve by POP);

2nd: stack can be used by user or by compiler for subroutine parameters (PUSH, POP in calling routine; addressed via offset calculation on stack pointer (SP) in called subroutine);

3rd: used by subroutine calls to store the program counter value for return at subroutine's end (RET);

4th: used by interrupt - system stores the actual PC value first, then the actual status register content (on top of stack) on return from interrupt (RETI) the system get the same status as just before the interrupt happened (as long as none has changed the value on TOS) and the same program counter value from stack.

R2: Status Register (SR)

The Status Register (SR/R2) stores the state and control bits. The system flags are changed automatically by the CPU depending on the result of an operation in a register. The reserved bits of the SR are used to support the constants generator. See the device-specific data sheets for more details.

SR

15	14	13	12	11	10	9	8	7	6	5	4	
Reserved for CG1							V	SCG1	SCG0	OSCOFF	CPUOFF	

Bit		Description
8	V	Overflow bit.V = 1 ⇒ Result of an arithmetic operation overflows the signed-variable

		range.
7	SCG1	System clock generator 0.SCG1 = 1 ⇒ DCO generator is turned off – if not used for MCLK or SMCLK.
6	SCG0	System clock generator 1.SCG0 = 1 ⇒ FLL+ loop control is turned off.
5	OSCOFF	Oscillator Off.OSCOFF = 1 ⇒ turns off LFXT1 when it is not used for MCLK or SMCLK.
4	CPUOFF	CPU off.CPUOFF = 1 ⇒ disable CPU core.
3	GIE	General interrupt enable.GIE = 1 ⇒ enables maskable interrupts.
2	N	Negative flag.N = 1 ⇒ result of a byte or word operation is negative.
1	Z	Zero flag.Z = 1 ⇒ result of a byte or word operation is 0.
0	C	Carry flag.C = 1 ⇒ result of a byte or word operation produced a carry.

R2/R3: Constant Generator Registers (CG1/CG2)

Depending of the source-register addressing modes (As) value, six commonly used constants can be generated without a code word or code memory access to retrieve them.

This is a very powerful feature, which allows the implementation of emulated instructions, for example, instead of implementing a core instruction for an increment, the constant generator is used.

Register	As	Constant	Remarks
R2	00	-	Register mode
R2	01	(0)	Absolute mode
R2	10	00004h	+4, bit processing
R2	11	00008h	+8, bit processing
R3	00	00000h	0, word processing
R3	01	00001h	+1
R3	10	00002h	+2, bit processing
R3	11	0FFFFh	-1, word processing

R4 - R15: General–Purpose Registers

These general-purpose registers are used to store data values, address pointers, or index values and can be accessed with byte or word instructions.

Request the MSP430 Teaching ROM Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

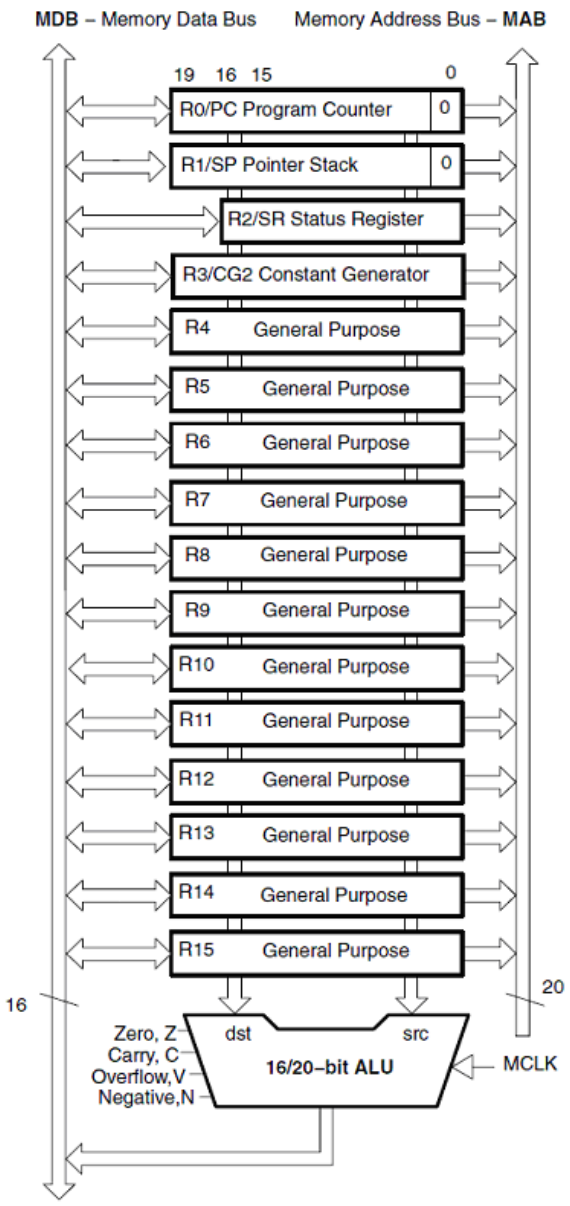
Central Processing Unit (MSP430X CPU)

Central Processing Unit (MSP430X CPU)

Main features of the MSP430X CPU architecture

The MSP430X CPU extends the addressing capabilities of the MSP430 family beyond 64 kB to 1 MB. To achieve this, there are some changes to the addressing modes and two new types of instructions. One type of new instructions allows access to the entire address space, and the other is designed for address calculations.

The MSP430X CPU address bus is 20 bits, but the data bus is still 16 bits. The CPU supports 8-bit, 16-bit and 20-bit memory accesses. Despite these changes, the MSP430X CPU remains compatible with the MSP430 CPU, having a similar number of registers. A block diagram of the MSP430X CPU is shown in the figure below:
MSP430X CPU block diagram.



Although the MSP430X CPU structure is similar to that of the MSP430 CPU, there are some differences that will now be discussed.

With the exception of the status register SR, all MSP430X registers are 20 bits. The CPU can now process 20-bit or 16-bit data.

MSP430X CPU registers

R0 (PC) - Program Counter

Has the same function as the MSP430 CPU, although now it has 20 bits.

R1 (SP) - Stack Pointer

Has the same function as the MSP430 CPU, although now it has 20 bits.

R2 (SR) - Status Register

Has the same function as the MSP430 CPU, but still only has 16 bits.
Description of the SR bits.

Bit	Description
Reserved	Reserved
V	Overflow bit. This bit is set when the result of an arithmetic operation overflows the signed-variable range. `ADD(.B), ADDX(.B,.A), ADDC(.B), ADDCX(.B.A), ADDA` — Set when: positive + positive = negative; negative + negative = positive; otherwise reset `SUB(.B), SUBX(.B,.A), SUBC(.B),SUBCX(.B,.A), SUBA, CMP(.B), CMPX(.B,.A), CMPA` — Set when: positive – negative = negative; negative – positive = positive; otherwise reset
SCG1	System clock generator 1. This bit, when set, turns off the DCO dc generator if DCOCLK is not used for MCLK or SMCLK.
SCG0	System clock generator 0. This bit, when set, turns off the FLL+ loop control.
OSCOFF	Oscillator Off. This bit, when set, turns off the LFXT1 crystal oscillator when LFXT1CLK is not used for MCLK or SMCLK.
CPUOFF	CPU off. This bit, when set, turns off the CPU.
GIE	General interrupt enable. This bit, when set, enables maskable interrupts. When reset, all maskable interrupts are disabled.
N	Negative bit. This bit is set when the result of an operation is negative and cleared when the result is positive.

Z	Zero bit. This bit is set when the result of an operation is zero and cleared when the result is not zero.
C	Carry bit. This bit is set when the result of an operation produced a carry and cleared when no carry occurred.

R2 (CG1) and R3 (CG2) - Constant Generators

The registers R2 and R3 can be used to generate six different constants commonly used in programming, without the need to add an extra 16-bit word of code to the instruction. The constants below are chosen based on the bit (As) of the instruction that selects the addressing mode.
Values of constant generators.

Register	As	Constant	Remarks
R2	00	-	Register mode
R2	01	(0)	Absolute address mode
R2	10	00004h	+4, bit processing
R2	11	00008h	+8, bit processing
R3	00	00000h	0, word processing
R3	01	00001h	+1
R3	10	00002h	+2, bit processing
R3	11	FFh, FFFFh, FFFFFh	-1, word processing

Whenever the operand is one of these six constants, the registers are selected automatically. Therefore, when used in constant mode, registers R2 and R3 cannot be addressed explicitly by acting as source registers.

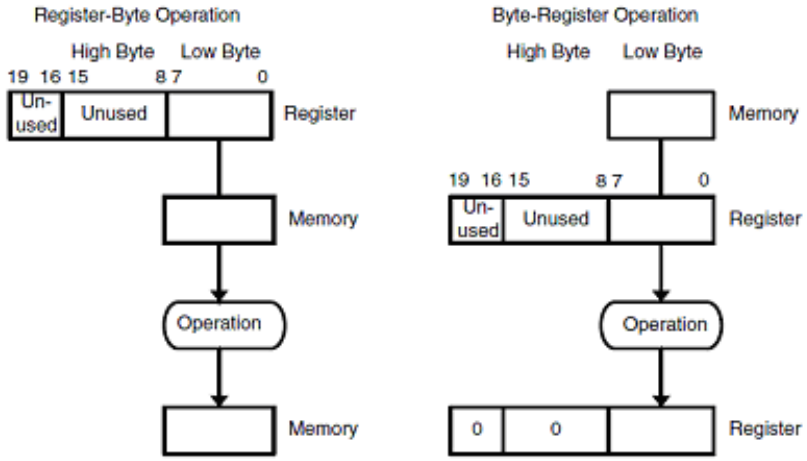
R4-R15 – General-purpose registers

These registers have the same function as the MSP430 CPU, although they now have 20 bits. They can store 8-bit, 16-bit or 20-bit data. Any byte written to one of these registers clears bits 19:8. Any word written to one of these registers clears bits 19:16. The exception to this rule is the instruction SXT, which extends the sign value to fill the 20-bit register.

The following figures illustrate how the operations are conducted for the exchange of information between memory and registers, for the following formats: byte (8 bits), word (16 bits) and address (20 bits).

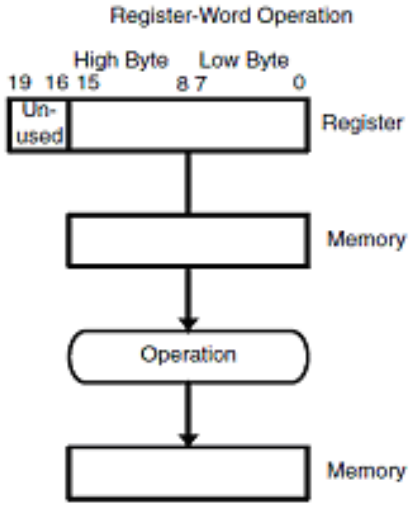
The following figure illustrates the handling of a byte (8 bits) using the suffix .B.

Example: Register-Byte/Byte-Register operation.

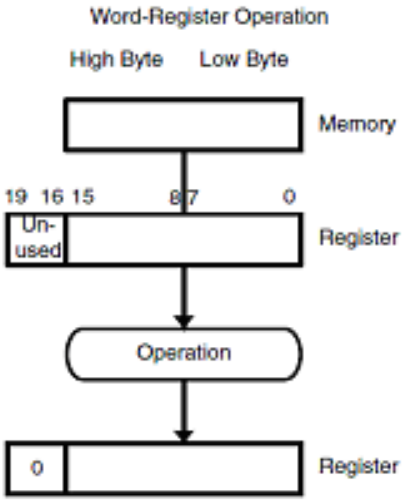


The following figure illustrates the handling of a word (16-bit) using the suffix .W.

Example: Register-Word/Word-Register operation.

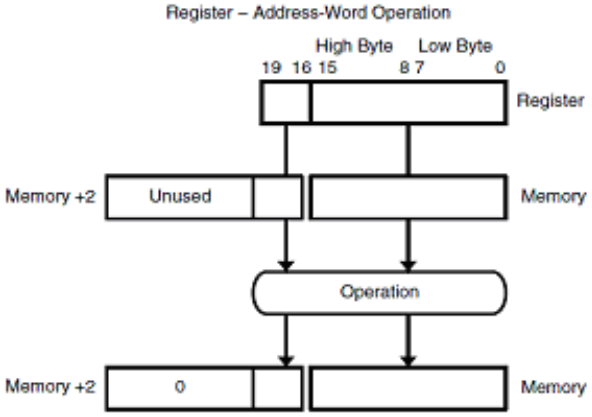


Example: Register-Word/Word-Register operation.

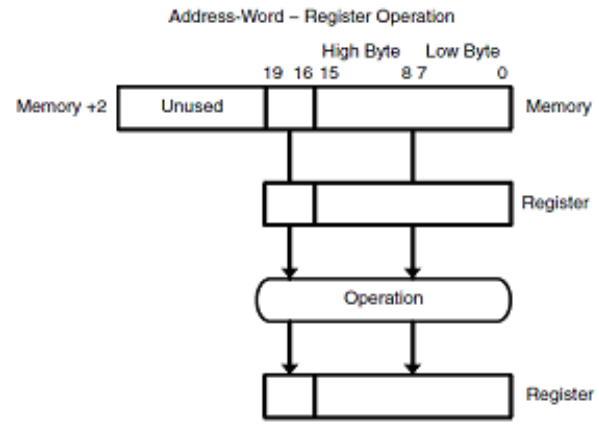


The following figure illustrates the manipulation of an address (20 bits) using the suffix .A.

Example: Register - Address-Word operation.



Example: Address-Word - Register operation.



Request the MSP430 Teaching ROM Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Addressing modes

Addressing modes

The MSP430 supports seven addressing modes for the source operand and four addressing modes for the destination operand (see bellow). The following sections describe each of the addressing modes, with a brief description, an example and the number of CPU clock cycles required for an instruction, depending on the instruction format and the addressing modes used.

Mode	Source operand	Destination operand	Description
Register mode	X	X	Single cycle
Indexed mode	X	X	Table processing
Symbolic mode	X	X	Easy to read code, PC relative
Absolute mode	X	X	Directly access any memory location
Indirect register mode	X		Access memory with pointers
Indirect auto increment mode	X		Table processing
Immediate	X		Unrestricted

mode			constant values

Before describing the addressing modes, it is important to mention the clock cycles required by interrupts and resets.

Action	Cycles	Length (words)
Return from interrupt	5	1
Interrupt accepted	6	-
Watchdog timer reset	4	-
Hard reset	4	-

Register Mode

Register mode operations work directly on the processor registers, R4 through R15, or on special function registers, such as the program counter or status register. They are very efficient in terms of both instruction speed and code space.

Description: Register contents are operands.

Source mode bits: As = 00 (source register defined in the opcode).

Destination mode bit: Ad=0 (destination register defined in the opcode).

Syntax: Rn.

Length: One or two words.

Comment: Valid for source and destination.

Example 1: Move (copy) the contents of source (register R4) to destination (register R5). Register R4 is not affected.

Before operation: R4=A002h R5=F50Ah PC = PC_{pos}

Operation: MOV R4, R5

After operation: R4=A002h R5=A002h PC = PC_{pos} + 2

The first operand is in register mode and depending on the second operand mode, the cycles required to complete an instruction will differ. The next table shows the cycles required to complete an instruction, depending on the second operand mode.

Operands	**2 nd operand mode**	Operator	Cycles	Length (words)
2	Register	Any	1*	1
2	Indexed, Symbolic or Absolute	Any	4	2
1	N/A	RRA, RRC, SWPB or SXT	1	1
1	N/A	PUSH	3	1

1	N/A	CALL	4	1

Indexed mode

The Indexed mode commands are formatted as X(Rn), where X is a constant and Rn is one of the CPU registers. The absolute memory location X+Rn is addressed. Indexed mode addressing is useful for applications such as lookup tables.

Description: (Rn + X) points to the operand. X is stored in the next word.

Source mode bits: As = 01 (memory location is defined by the word immediately following the opcode).

Destination mode bit: Ad=1 (memory location is defined by the word immediately following the opcode).

Syntax: X(Rn).

Length: Two or three words.

Comment: Valid for source and destination.

Example 2: Move (copy) the contents at source address (F000h + R5) to destination (register R4).

Before operation: R4=A002h R5=050Ah Loc:0xF50A=0123h

Operation: MOV F000h(R5), R4

After operation: R4=0123h R5=050Ah Loc:0xF50A=0123h

Operands	**2 nd**	Operator	Cycles	Length

	operand mode			(words)
2	Register	Any	3	2
2	Indexed, Symbolic or Absolute	Any	6	3
1	N/A	RRA, RRC, SWPB or SXT	4	2
1	N/A	CALL or PUSH	5	2

Symbolic mode

Symbolic mode allows the assignment of labels to fixed memory locations, so that those locations can be addressed. This is useful for the development of embedded programs.

Description: (PC + X) points to the operand. X is stored in the next word. Indexed mode X(PC) is used.

Source mode bits: As = 01 (memory location is defined by the word immediately following the opcode).

Destination mode bit: Ad=1 (memory location is defined by the word immediately following the opcode).

Syntax: ADDR.

Length: Two or three words.

Comment: Valid for source and destination.

Example 3: Move the content of source address XPT (x pointer) to the destination address YPT (y pointer).

Before operation: XPT=A002h Location YPT=050Ah

Operation: MOV XPT, YPT

After operation: XPT= A002h Location YPT=A002h

Operands	**2 nd operand mode**	Operator	Cycles	Length (words)
2	Register	Any	3	2
2	Indexed, Symbolic or Absolute	Any	6	3
1	N/A	RRA, RRC, SWPB or SXT	4	2
1	N/A	CALL or PUSH	5	2

Absolute mode

Similar to Symbolic mode, with the difference that the label is preceded by “&”.

Description: The word following the instruction contains the absolute address. X is stored in the next word. Indexed mode X(SR) is used.

Source mode bits: As = 01 (memory location is defined by the word immediately following the opcode).

Destination mode bit: Ad=1 (memory location is defined by the word immediately following the opcode).

Syntax: &ADDR.

Length: Two or three words.

Comment: Valid for source and destination.

Example 4: Move the content of source address XPT to the destination address YPT.

Before operation: Location XPT=A002h Location YPT=050Ah

Operation: MOV &XPT, &YPT

After operation: Location XPT= A002h Location YPT=A002h

Operands	**2 nd operand mode**	Operator	Cycles	Length (words)
2	Register	Any	3	2
2	Indexed, Symbolic or Absolute	Any	6	3

1	N/A	RRA, RRC, SWPB or SXT	4	2
1	N/A	CALL or PUSH	5	2

Indirect register mode

The data word addressed is located in the memory location pointed to by Rn. Indirect mode is not valid for destination operands, but can be emulated with the indexed mode format 0(Rn).

Description: Rn is used as a pointer to the operand.

Source mode bits: As = 10.

Syntax: @Rn.

Length: One or two words.

Comment: Valid only for source operand. The substitute for destination operand is 0(Rn).

Example 5: Move the contents of the source address (contents of R4) to the destination (register R5). Register R4 is not modified.

Before operation: R4=A002h R5=050Ah Loc:0xA002=0123h

Operation: MOV @(R4), R5

After operation: R4= A002h R5=0123h Loc:0xA002=0123h

Operands	**2 nd operand mode**	Operator	Cycles	Length (words)
2	Register	Any	2*	1
2	Indexed, Symbolic or Absolute	Any	5	2
1	N/A	RRA, RRC, SWPB or SXT	3	1
1	N/A	CALL or PUSH	4	1

Indirect auto increment mode

Similar to indirect register mode, but with indirect auto increment mode, the operand is incremented as part of the instruction. The format for operands is @Rn+. This is useful for working on blocks of data.

Description: Rn is used as a pointer to the operand. Rn is incremented afterwards by 1 for byte instructions and by 2 for word instructions.

Source mode bits: As = 11.

Syntax: @Rn+.

Length: One or two words.

Comment: Valid only for source operand. The substitute for destination operand is 0(Rn) plus second instruction INCD Rn.

Example 6: Move the contents of the source address (contents of R4) to the destination (register R5), then increment the value in register R4 to point to the next word.

Before operation: R4=A002h R5=050Ah Loc:0xA002=0123h

Operation: MOV @R4+, R5

After operation: R4= A004h R5=0123h Loc:0xA002=0123h

Operands	**2 nd operand mode**	Operator	Cycles	Length (words)
2	Register	Any	2*	1
2	Indexed, Symbolic or Absolute	Any	5	2
1	N/A	RRA, RRC, SWPB or SXT	3	1
1	N/A	PUSH	4	1
1	N/A	CALL	5	1

Immediate mode

Immediate mode is used to assign constant values to registers or memory locations.

Description: The word following the instruction contains the immediate constant N. Indirect autoincrement mode @PC+ is used.

Source mode bits: As = 11.

Syntax: #N.

Length: Two or three words. It is one word less if a constant in CG1 or CG2 can be used.

Comment: Valid only for source operand.

Example 7: Move the immediate constant E2h to the destination (register R5).

Before operation: R4=A002h R5=050Ah

Operation: MOV #E2h, R5

After operation: R4= A002h R5=00E2h

Operands	**2 nd operand mode**	Operator	Cycles	Length (words)
2	Register	Any	2*	2
2	Indexed, Symbolic or Absolute	Any	5	3

1	N/A	RRA, RRC, SWPB or SXT	N/A	N/A
1	N/A	PUSH	4	2
1	N/A	CALL	5	2

Request the MSP430 Teaching ROM Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

MSP430 instruction set

Instruction set

The MSP430 instruction set consists of 27 core instructions. Additionally, it supports 24 emulated instructions. The core instructions have unique op-codes decoded by the CPU, while the emulated ones need assemblers and compilers to generate their mnemonics.

There are three core-instruction formats:

- Double operand;

- Single operand;

- Program flow control - Jump.

Byte, word and address instructions are accessed using the .B, .W or .A extensions. If the extension is omitted, the instruction is interpreted as a word instruction.

Double operand instructions

The double operand instruction is formatted as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode				S-Reg				Ad	B/W	As		D-Reg			

Bit		Description
15-12	opcode	
11-8	S-Reg	The working register used for the source operand (src)
7	Ad	The addressing bits responsible for the addressing mode used for the destination operand (dst)
6	B/W	Byte or word operation:B/W=0: word operation; B/W=1: byte operation
5-4	As	The addressing bits responsible for the addressing mode used for the source operand (src)
3-0	D-Reg	The working register used for the destination operand (dst)

The next tableshows the double operand instructions that are not emulated.

Mnemonic	Operation	Description
Arithmetic instructions		
ADD(.B or .W) src,dst	src+dst→dst	Add source to destination
ADDC(.B or .W) src,dst	src+dst+C→dst	Add source and carry to destination
DADD(.B or .W) src,dst	src+dst+C→dst (dec)	Decimal add source and carry to destination
SUB(.B or .W) src,dst	dst+.not.src+1→dst	Subtract source from destination
SUBC(.B or .W) src,dst	dst+.not.src+C→dst	Subtract source and not carry from destination
Logical and register control instructions		
AND(.B or .W) src,dst	src.and.dst→dst	AND source with destination
BIC(.B or .W) src,dst	.not.src.and.dst→dst	Clear bits in destination
BIS(.B or .W) src,dst	src.or.dst→dst	Set bits in destination
BIT(.B or .W) src,dst	src.and.dst	Test bits in destination
XOR(.B or .W) src,dst	src.xor.dst→dst	XOR source with destination
Data instructions		
CMP(.B or .W) src,dst	dst-src	Compare source to destination
MOV(.B or .W) src,dst	src→dst	Move source to destination

Depending on the double operand instruction result, the status bits may be affected. The following gives the conditions for setting and resetting the status bits.

	Status bits			
Mnemonic	V	N	Z	C
Arithmetic instructions				
ADD(.B or .W) src,dst	=1, Arithmetic overflow=0, otherwise	=1, negative result=0, if positive	=1, null result=0, otherwise	=1, carry from result=0, if not
ADDC(.B or .W) src,dst	=1, Arithmetic overflow=0, otherwise	=1, negative result=0, if positive	=1, null result=0, otherwise	=1, carry from MSB result=0, if not
DADD(.B	-	=1, MSB=1=0,	=1, null	=1, result > 99(99)

or .W) src,dst		otherwise	result=0, otherwise	
SUB(.B or .W) src,dst	=1, Arithmetic overflow=0, otherwise	=1, negative result=0, if positive	=1, null result=0, otherwise	=1, if no borrow=0, otherwise
SUBC(.B or .W) src,dst	=1, Arithmetic overflow=0, otherwise	=1, negative result=0, if positive	=1, null result=0, otherwise	=1, if no borrow=0, otherwise
Logical and register control instructions				
AND(.B or .W) src,dst	=0	=1, MSB result set=0, if not set	=1, null result=0, otherwise	=1, not zero=0, otherwise
BIC(.B or .W) src,dst	-	-	-	-
BIS(.B or .W) src,dst	-	-	-	-
BIT(.B or .W) src,dst	=0	=1, MSB result set=0, otherwise	=1, null result=0, otherwise	=1, not zero=0, otherwise
XOR(.B or .W) src,dst	=1, both operands negative	=1, MSB result set=0, otherwise	=1, null result,=0, otherwise	=1, not zero=0, otherwise
Data instructions				
CMP(.B or .W) src,dst	=1, Arithmetic overflow=0, otherwise	=1, src>=dst=0, src<dst	=1, src=dst=0, otherwise	=1, carry from MSB result=0, if not
MOV(.B or .W) src,dst	-	-	-	-

Single operand instructions

The single operand instruction is formatted as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode									B/W	Ad		D/S-Reg			

Bit		Description
15-7	opcode	
6	B/W	Byte or word operation:B/W=0: word operation; B/W=1: byte operation
5-4	Ad	The addressing bits responsible for the addressing mode used for the source operand (src)
3-0	D/S-Reg	The working register used for the destination operand (dst) or for the source operand (src)

The next table shows the single operand instructions that are not emulated.

Mnemonic	Operation	Description
Logical and register control instructions		
RRA(.B or .W) dst	MSB→MSB→…LSB→C	Roll destination right
RRC(.B or .W) dst	C→MSB→…LSB→C	Roll destination right through (from) carry
SWPB(.B or .W) dst	Swap bytes	Swap bytes in destination
SXT dst	bit 7→bit 8…bit 15	Sign extend destination
PUSH(.B or .W) src	SP-2→SP, src→@SP	Push source to stack
Program flow control instructions		
CALL(.B or .W) dst	SP-2→SP, PC+2→@SPdst→PC	Subroutine call to destination
RETI	TOS→SR, SP+2→SPTOS→PC, SP+2→SP	Return from interrupt

Conditions for status bits, depending on the single operand instruction result.

	Status bits			

Mnemonic	V	N	Z	C
Logical and register control instructions				
RRA(.B or .W) dst	=0	=1, negative result=0, otherwise	=1, null result,=0, otherwise	Loaded from LSB
RRC(.B or .W) dst	=1, dst positive & C=1=0, otherwise	=1, negative result=0, otherwise	=1, null result,=0, otherwise	Loaded from LSB
SWPB(.B or .W) dst	-	-	-	-
SXT dst	=0	=1, negative result=0, otherwise	=1, null result,=0, otherwise	=1, not zero=0, otherwise
PUSH(.B or .W) src	-	-	-	-
Data instructions				
CALL(.B or .W) dst	-	-	--	-
RETI	restored from stack	restored from stack	restored from stack	restored from stack

Program flow control - Jumps

The jump instruction is formatted as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode			C			10 bit PC offset									

Bit		Description
15-13	opcode	
12-10	C	
9-0	PC offset	PCnew = PCold + 2 + PCoffset × 2

The following table shows the program flow control (jump) instructions that are not emulated.

Mnemonic	Description
Program flow control instructions	
JEQ/JZ label	Jump to label if zero flag is set
JNE/JNZ label	Jump to label if zero flag is reset
JC label	Jump to label if carry flag is set
JNC label	Jump to label if carry flag is reset
JN label	Jump to label if negative flag is set
JGE label	Jump to label if greater than or equal
JL label	Jump to label if less than
JMP label	Jump to label unconditionally

Emulated instructions

The next gives the different emulated instructions. This table also contains the type of operation and the emulated instruction based on the core instructions.

Mnemonic	Operation	Emulation	Description
Arithmetic instructions			
ADC(.B or .W) dst	dst+C→dst	ADDC(.B or .W) #0,dst	Add carry to destination
DADC(.B or .W) dst	dst+C→dst (decimally)	DADD(.B or .W) #0,dst	Decimal add carry to destination
DEC(.B or .W) dst	dst-1→dst	SUB(.B or .W) #1,dst	Decrement destination
DECD(.B or .W) dst	dst-2→dst	SUB(.B or .W) #2,dst	Decrement destination twice
INC(.B or	dst+1→dst	ADD(.B or .W)	Increment destination

.W) dst		#1,dst	
INCD(.B or .W) dst	dst+2→dst	ADD(.B or .W) #2,dst	Increment destination twice
SBC(.B or .W) dst	dst+0FFFFh+C→dstdst+0FFh→dst	SUBC(.B or .W) #0,dst	Subtract source and borrow /.NOT. carry from dest.
Logical and register control instructions			
INV(.B or .W) dst	.NOT.dst→dst	XOR(.B or .W) #0(FF)FFh,dst	Invert bits in destination
RLA(.B or .W) dst	C MSB MSB-1…LSB+1 LSB 0	ADD(.B or .W) dst,dst	Rotate left arithmetically
RLC(.B or .W) dst	C MSB MSB-1…LSB+1 LSB C	ADDC(.B or .W) dst,dst	Rotate left through carry
Data instructions			
CLR(.B or .W) dst	0→dst	MOV(.B or .W) #0,dst	Clear destination
CLRC	0→C	BIC #1,SR	Clear carry flag
CLRN	0→N	BIC #4,SR	Clear negative flag
CLRZ	0→Z	BIC #2,SR	Clear zero flag
POP(.B or .W) dst	@SP→tempSP+2→SPtemp→dst	MOV(.B or .W) @SP+,dst	Pop byte/word from stack to destination
SETC	1→C	BIS #1,SR	Set carry flag
SETN	1→N	BIS #4,SR	Set negative flag
SETZ	1→Z	BIS #2,SR	Set zero flag
TST(.B or .W) dst	dst + 0FFFFh + 1dst + 0FFh + 1	CMP(.B or .W) #0,dst	Test destination
Program flow control			
BR dst	dst→PC	MOV dst,PC	Branch to destination
DINT	0→GIE	BIC #8,SR	Disable (general) interrupts
EINT	1→GIE	BIS #8,SR	Enable (general) interrupts
NOP	None	MOV #0,R3	No operation
RET	@SP→PCSP+2→SP	MOV @SP+,PC	Return from subroutine

Conditions for status bits, depending on the emulated instruction result.

	Status bits			
Mnemonic	V	N	Z	C
Arithmetic instructions				
ADC(.B or .W) dst	=1, Arithmetic overflow=0, otherwise	=1, negative result=0, if positive	=1, null result=0, otherwise	=1, dst from 0FFFFh to 0000=0, otherwise
DADC(.B or .W) dst	-	=1, MSB=1=0, otherwise	=1, dst=0=0, otherwise	=1, dst from 99(99) to 00(00)=0, otherwise
DEC(.B or .W) dst	=1, Arithmetic overflow=0, otherwise	=1, negative result=0, if positive	=1, dst contained 1=0, otherwise	=1, dst contained 0=0, otherwise
DECD(.B or .W) dst	=1, Arithmetic overflow=0, otherwise	=1, negative result=0, if positive	=1, dst contained 2=0, otherwise	=1, dst contained 0 or 1=0, otherwise
INC(.B or .W) dst	=1, dst contained 07(FF)h=0, otherwise	=1, negative result=0, if positive	=1, dst contained FF(FF)h=0, otherwise	=1, dst contained FF(FF)h=0, otherwise
INCD(.B or .W) dst	=1, dst contained 07(FFE)h=0, otherwise	=1, negative result=0, if positive	=1, dst contained FF(FE)h=0, otherwise	=1, dst contained FF(FF)h or FF(FE)h=0, otherwise
SBC(.B or .W) dst	=1, Arithmetic overflow=0, otherwise	=1, negative result=0, if positive	=1, null result,=0, otherwise	=1, if no borrow=0, otherwise
Logical and register control instructions				
INV(.B or .W) dst	=1, negative initial dst=0, otherwise	=1, negative result=0, if positive	=1, dst contained FF(FF)h=0, otherwise	=1, not zero=0, otherwise
RLA(.B or .W) dst	=1, Arithmetic overflow=0, otherwise	=1, negative result=0, if positive	=1, null result,=0, otherwise	Loaded from MSB
RLC(.B or .W) dst	=1, Arithmetic overflow=0, otherwise	=1, negative result=0, if positive	=1, null result,=0, otherwise	Loaded from MSB
Data instructions				

CLR(.B or .W) dst	-	-	-	-
CLRC	-	-	-	=0
CLRN	-	=0	-	-
CLRZ	-	-	=0	-
POP(.B or .W) dst	-	-	-	-
SETC	-	-	-	=1
SETN	-	=1	-	-
SETZ	-	-	=1	-
TST(.B or .W) dst	=0	=1, dst negative=0, otherwise	=1, dst contains zero=0, otherwise	=1
Program flow control				
BR dst	-	-	-	-
DINT	-	-	-	-
EINT	-	-	-	-
NOP	-	-	-	-
RET	-	-	-	-

Request the MSP430 Teaching ROM Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Code Composer Essentials

TI recently launched Code Composer Essentials v3. This IDE's latest version (version 3) supports all available MSP430 devices.

The new features of CCE v3 include:

- Free 16 kB code-limited version;

- Supports the large memory model (Place data >64k);

- Enhanced Compatibility with IAR C-code:

- #pragma (ISR declarations), most intrinsics.

- GDB Debugger replaced by TI proprietary debugger that allows faster single stepping;

- Hardware Multiplier libraries (16-bit and 32-bit multiplies);

- CCE v2 project support (auto convert);

- Breakpoints:

- Extended Emulation Module (EEM) support via unified breakpoint manager;

- Using of EEM (predefined Use Cases);

- Unlimited Breakpoints

Eclipse is a software development platform, developed in Java, which allows it to be used on different operating systems. One of its main features is that it is fully based on plug-ins, which gives it great versatility. Code Composer Essentials (CCE) version 3 is based on Eclipse release 3.2 (Callisto). On the market there are hundreds of plug-ins that can be added in

order to enhance or optimize a particular aspect of CCE. One of the available repositories for plug-ins developed for the Eclipse is the Eclipse Plugin Central located at http://www.eclipseplugincentral.com/. This development tool has advanced capabilities to support the development of applications for the MSP430 family. Among them are the support for the use of breakpoints, either hardware or software. CCE supports code debugging activities, with support for features such as code step-by-step execution, or fast and efficient access to registers and memory locations. There is complete compatibility between the C programming language syntax used and the great diversity of code examples available.

In addition, others plug-ins are also part of the default version, the important ones being:

- Concurrent Version System (CVS): For control of code versions in production.

- Plug-in Development Environment (PDE): Relevant for those who want to expand the functionality of IDE through plug-ins.

- JUnit: Framework for code validation and test

CCE installation

Most of the installation of CCE is automated. It is only necessary to provide some user indications as to how the program installation should continue.

Request the MSP430 Teaching ROM Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Introduction to CCE IDE

Introduction to CCE IDE

The introductory overview in the use of CCE will continue with a practical example, addressing some of its main features. Let us begin by building a project. This project will be configured with respect to the hardware, i.e., the MSP430 family device.

Launching the workbench

The term “Workbench” refers to the integrated development environment of all tools necessary for the development and management of projects. When CCE is started, it asks the user where they want to locate the work directory (workspace).

After choosing the location where the workspace will be stored, it opens by default in the project construction perspective. The concept associated with a perspective is important for the correct understanding of CCE operation. A perspective provides that for a given task there is an organization of windows most appropriate to its implementation. Changing perspective involves reformulating the workspace for a new Windows configuration that promotes the development of particular task. There are two major perspectives: **C/C++** for editing, management and compilation of projects, and **Debug** for debugging the applications. The working perspective is selected in the upper right hand corner of the application.

By default, the windows included in the **C/C++** perspective are: **C/C++ Projects** (to manage the projects); **Editor** (to edit files); **Outline** (to view data); **Console** (to send messages); **Problems** (identifies problems found in the project). The icons associated with the various tasks that can be performed in this perspective are shown together in *Table 1*.

Button	Description	Button	Description
	Open a new perspective		Save the active editor contents
	Save the contents of all editors		Save editor contents under a new name or location
	Opens the search dialog		Print editor contents
	Open a resource creation wizard (New)		Open a file creation wizard
	Open a folder creation wizard		Open a project creation wizard
	Open the import wizard		Open the export wizard
	Run incremental build (Build All)		Run a program
	Debug a program		Run an external tool
	Cut selection to clipboard		Copy selection to clipboard
	Paste selection from clipboard		Undo most recent edit

	Redo most recent undone edit		Navigate to next item in a list
	Navigate to previous item in a list		Navigate forwards
	Navigate backwards		Navigate up one level
	Add bookmark or task		Open a view's drop down menu
	Close view or editor		Pin editor to prevent automatic reuse
	Filter tasks or properties		Go to a task, problem, or bookmark in the editor
	Restore default properties		Show items as a tree
	Refresh view contents		Sort list in alphabetical order
	Cancel a running operation		Delete selected item or content
	Last edit location		Toggle Mark Occurrences
	Assembly		

	instruction only		

By default, the windows included in the **Debug** perspective are: **Debug** (provides information concerning the debug process); **Editor** (to edit files); **Variables/Expressions** (to evaluate variables and expressions values during debug); **Console** (console to send messages); **Registers/Breakpoints** (to evaluate the contents of registers and to define code breakpoint); and **Disassembly/Memory** (to evaluate the assembly code and memory map occupation). The icons associated with the various tasks that can be performed in this perspective are shown together in *Table 2*.

Icon	Command	Description
	Create New	Create a new project, folder, or file.
	Save	Save the content of the current editor. Disabled if the editor does not contain unsaved changes.
	Print	Prints the content of the current editor.
	Build All	Compiles all files for all projects in workbench.
	Enable/Disable Breakpoints	Enables or disables a breakpoint at the specified location.

	Toggle Breakpoint	Toggles a breakpoint at a specific address selected in the Edit window.
	Change Build Configuration	Lists available build configurations to choose.
	New C/C++ Project	Creates a new C/C++ project.
	New C/C++ Source Folder	Creates a source folder within the current project.
	New C/C++ Source File	Creates a source file within the current project.
	New C/C++ Class	Creates a C++ class within the current project.
	Debug Active Project	Debugs the current active project.
	Launch TI Debugger	Launches the TI specific debugger.
	Debug	Launches the Debug dialog box.
	Run	Launches the Run dialog box
	External Tools	Launches the External Tools dialog box

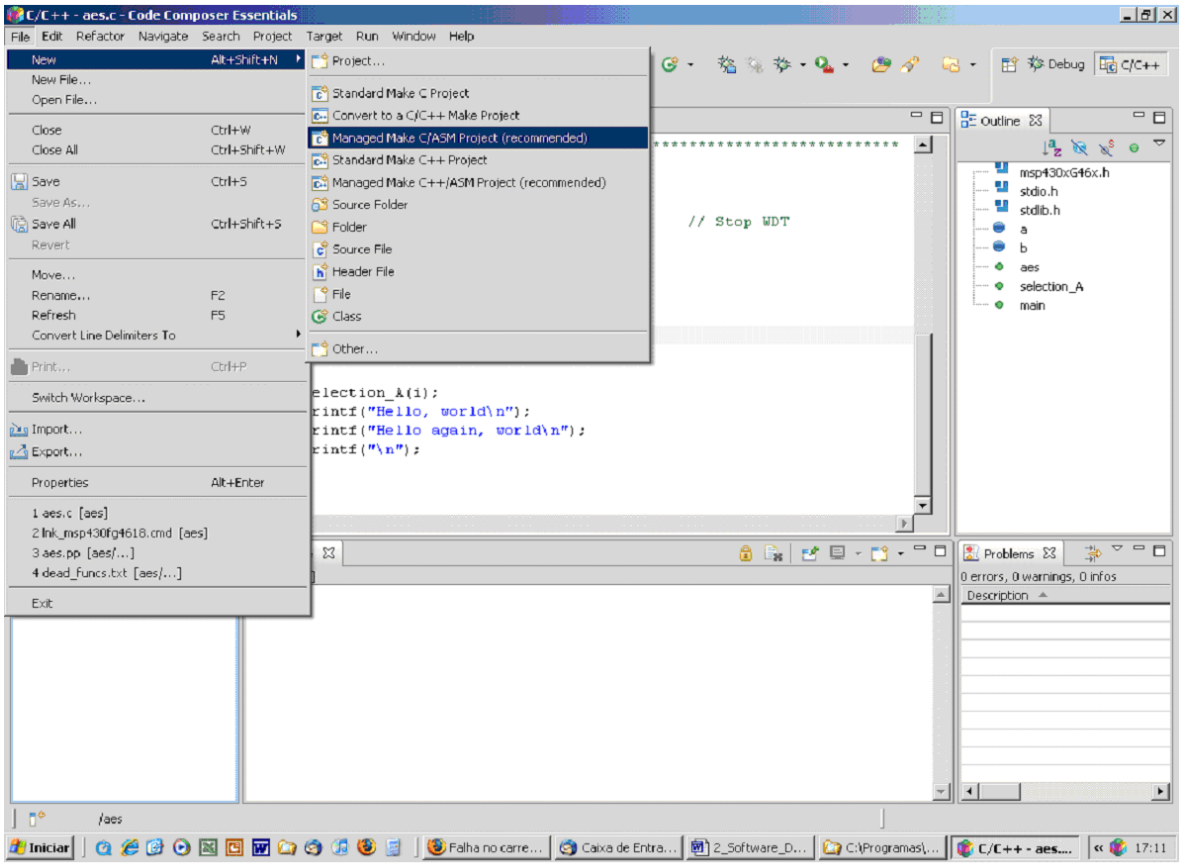
	Open Type	Brings up the Open Type selection dialog to open a type in the editor. The Open Type selection dialog shows all types existing in the workspace.
	Search	Launches the C/C++ Search dialog box
	Select Working Sets	Selects a working set from the list to be the active one. Working sets group elements for display in views or for operations on a set of elements.
	Next Annotation	Selects the next annotation in the resource that is currently active in the editor area. Supported in the Java editor.
	Previous Annotation	Selects the previous annotation in the resource that is currently active in the editor area. Supported in the Java editor.
	Go to Last Edit Location	Returns editor view to the last line edited, if the file that was last edited was closed it will be re-opened.
	Back	Navigates back through open files.
	Forward	Navigates forward through open files.

Request the MSP430 Teaching ROM Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Creating a Project

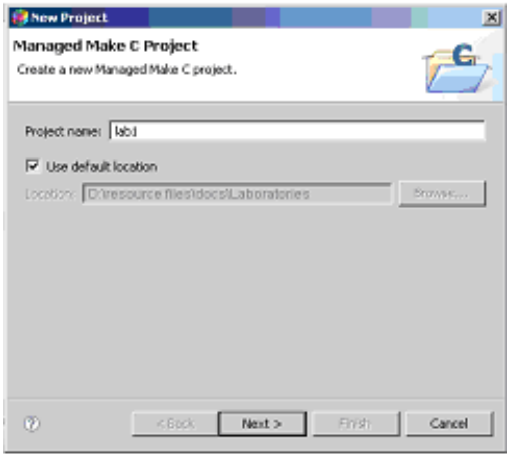
Select the option **File> New > Managed Make C\ASM Project** (recommended) to create a project. Other project options are available, but with the above option, the project process creation is more automated. The *Figure 1* shows the window where the option should be selected.

CCE workbench – Project creation process window.



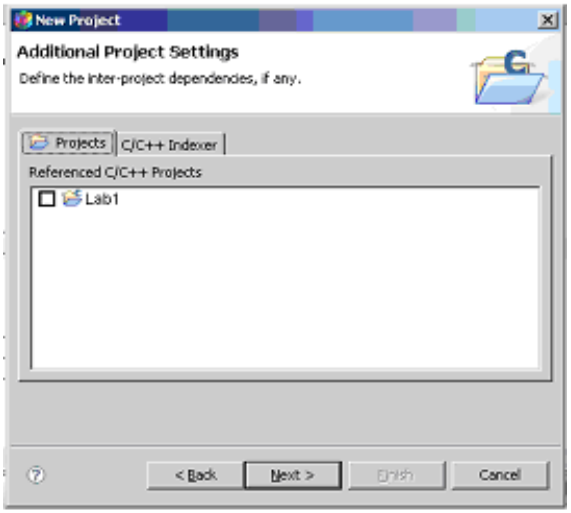
After choosing this option, a procedure for the creation of projects for the MSP430 family of microcontrollers is provided. The user must answer the first question concerning the project's name. By default, all the project files are stored within a folder, with the name of the project in the directory chosen for the *workspace*. The New Project window is shown in the *Figure 2*.

CCE workbench – New project name window.



Afterwards, some additional settings are made to the project, such as whether there is any dependency of this project on another. If this condition is true, the dependency should be established through the window shown in *Figure 3*.

CCE workbench – Project dependency window.



Information indexing functionality is part of the C/C++ Project. It uses a parser to create a database of the contents of the project files. This feature is used during the information search, the project navigation, and in the content assistant. The indexing task is performed in the background and reacts to changes in content such as: C/C++ project creation or deletion; file creation or deletion; file import; content of files changes.

There are three options for setting up the operation of this functionality:

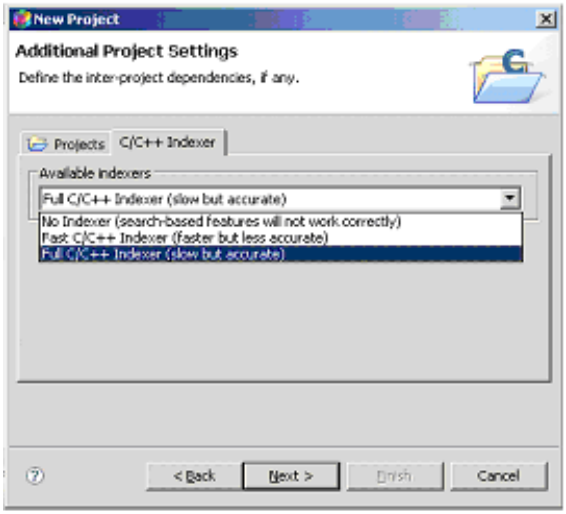
- Without Project contents indexing (No Indexer);

- Fast C/C++ or;

- Full Indexer C/C++ Indexer).

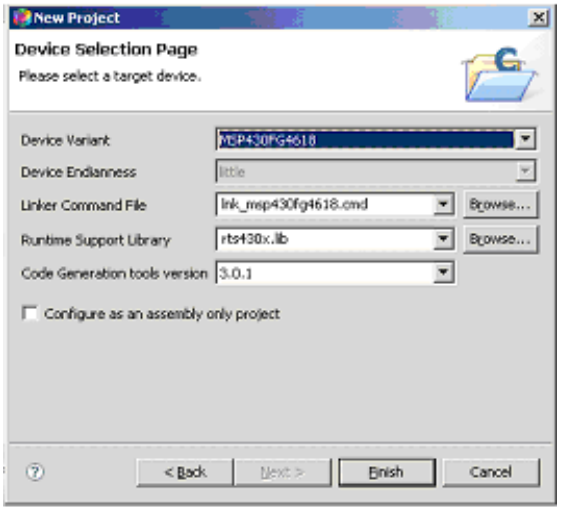
These two last options differ mainly in the required processing time of the indexing task and results quality. The configuration window of this feature is displayed in *Figure 3*.

CCE workbench – Project indexing window.



In the final window displayed during the project's creation procedure (see *Figure 4*), the device with which the project is being developed must be chosen. By choosing the device, the appropriate linker command file and supporting libraries are selected automatically.

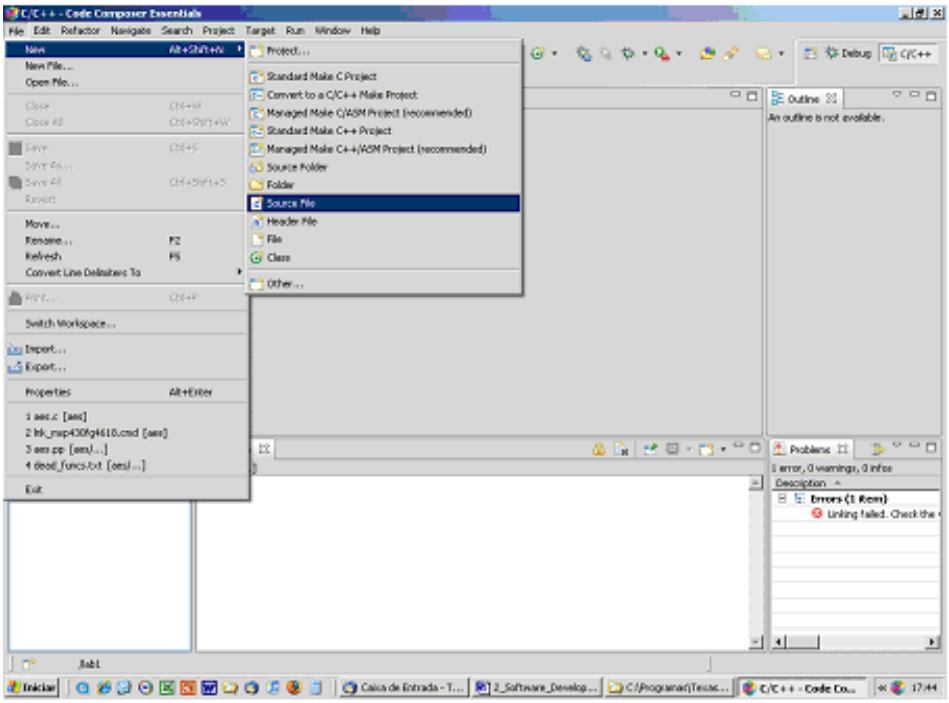
CCE workbench – Device selection window.



The project's creation can then be finalised by choosing the option **Finish**. At any time, it is possible to go back to previous windows by choosing the option **Back**.

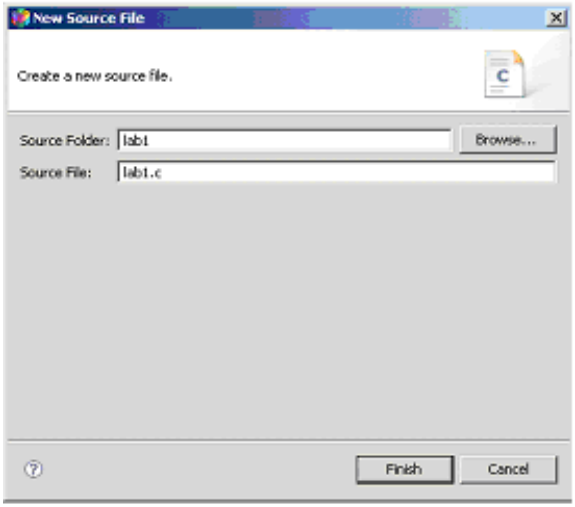
The next step is to add the source code file to the project. Choose **File > New > Source File**. In this menu the option to create *.C* type file should be selected, as shown in *Figure 5*.

CCE workbench – Source code file creation procedure.



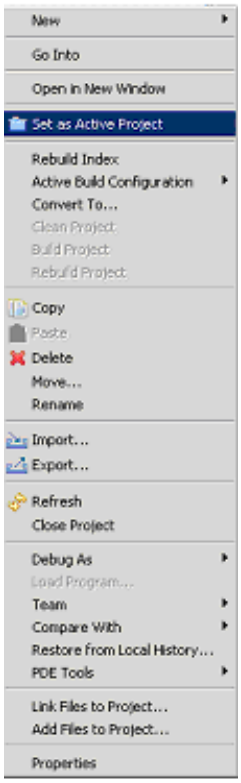
The name of the file is then requested in the window as shown in *Figure 6*. Do not forget to add the file extension such as “myfile.c” so that it is recognized as a C file.

CCE workbench – Source code file creation window.



The project is automatically selected as the default project. Although the workspace allows several projects to be opened simultaneously, it allows only one of them to be active. To select an active project, its name must be selected with the mouse’s right button in **C/C++ Project** view, in order to show the context menu. Then the option **set as active project** must be selected. From here, the expression [**Active-Debug**] will appear. In the context menu there are other options to manage the project: add or remove files, import or export resources, edit the properties and so on.

CCE workbench – Set as an active project window.



At this point, it is possible to start editing the project's source code. CCE has all the capabilities inherited from the Eclipse edition. Adding the file lab1.c, which already exists, is done through the option **add file to project**. This file can be found in **Project > add file to project**, as in the context menu of the view **C/C++ Projects**. The file lab1.c can be removed from the project by simply selecting it in the view and selecting the option **delete**. Note that when the file is removed, it will be cleared from the directory.

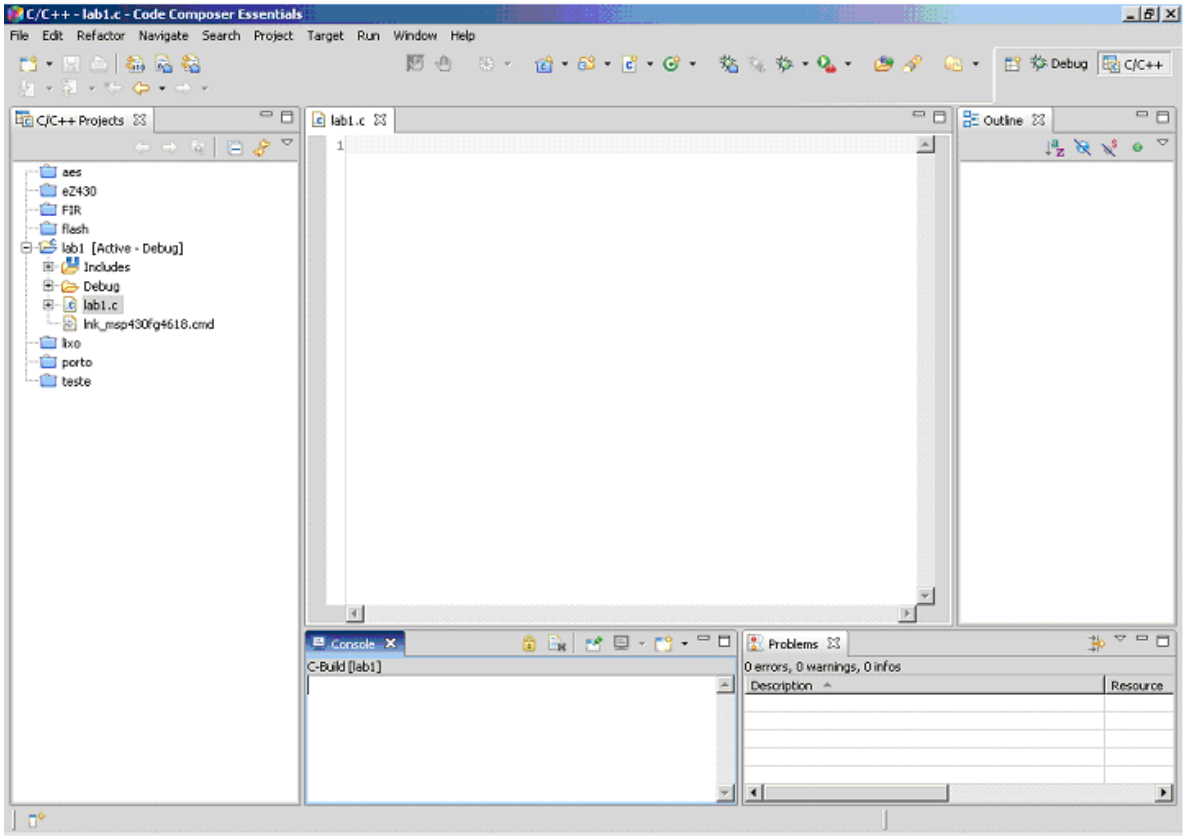
Request the MSP430 Teaching ROM Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Code Editor

Code Editor

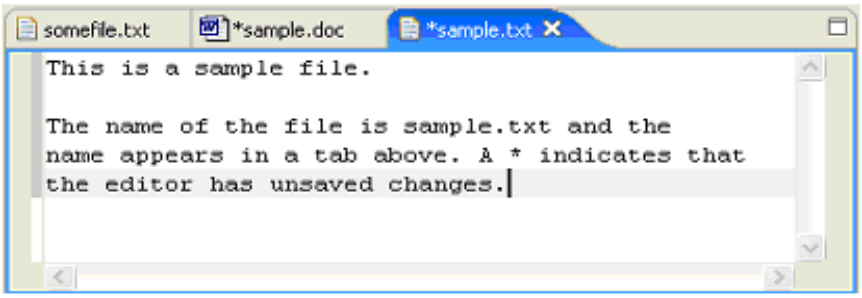
The text editor included in CCE is a versatile tool and very effective for helping with the code editing task. The C/C++ perspective is shown in *Figure 1*.

CCE workbench – C/C++ perspective.



The text editor has a set of features that allow speeding up the code editing process. An overview of the text editor is shown in *Figure 2*.

CCE workbench – Text editor window.

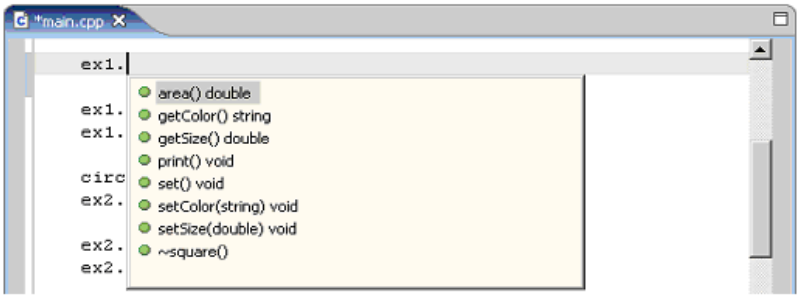


Code editing is greatly facilitated using features such as search and replace. To accomplish this task, the user must select **Edit > Find/Replace**. In addition to the normal features of search and replace, the option **Search > File** allows the use of more elaborate expressions. For example, it provides the global replacement in all files of a specific directory. The search and replace tasks previously performed can be found on the **drop-list**.

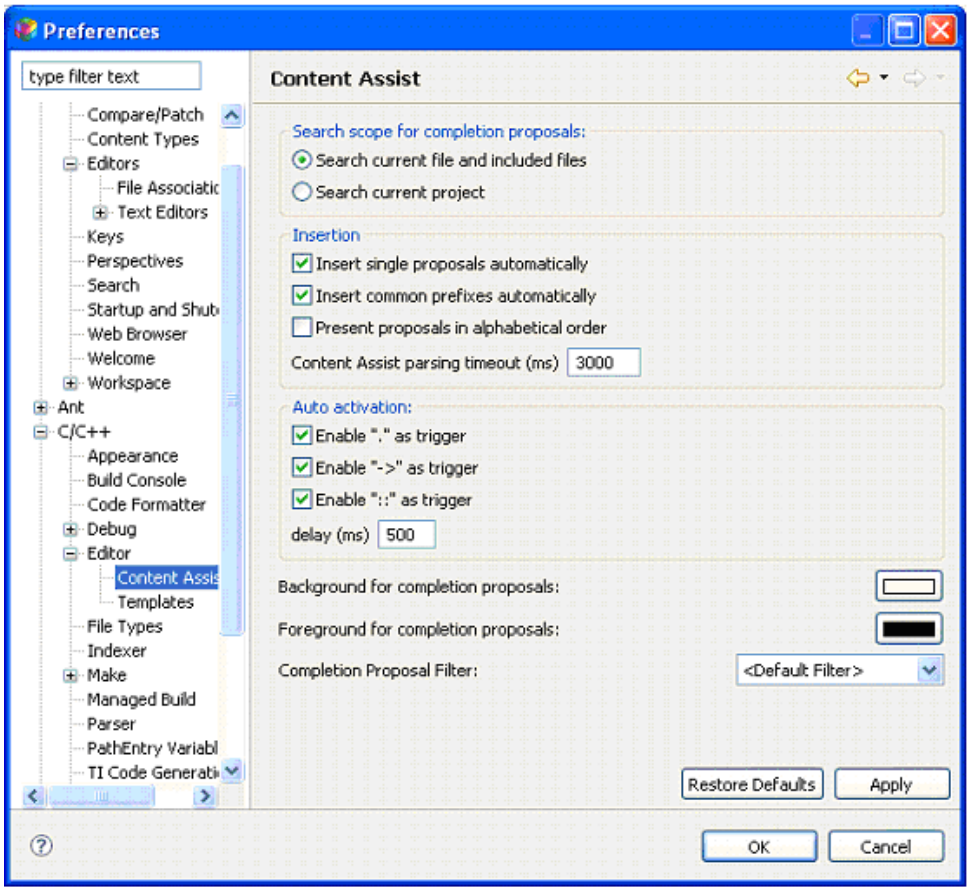
CCE can regularly save the opened files for editing in order to prevent losses caused by system crashes. To use this function, select **Window > Preferences > General > Workspace** and specify the time interval at which this task should be performed automatically. The project can also be saved whenever it goes through project build.

The content wizard is a very effective tool to support the writing of code. It is possible to automatically insert a code structure model, previously defined, as an alternative to writing it out completely (see an example in *Figure 3*). To insert a model of a structure, it is only necessary to write the first letters in the text editor and then press the *Ctrl + Space* keys in order to display a list of the corresponding models. The options in the list can be reduced by continuing writing the structure name. The *Arrow Up* and *Arrow Down* keys can be used to select the desired model and by pressing the *Enter* key to accept the selection. At any time the *Esc* key allows editing to continue without the use of the content wizard.

CCE workbench – Content wizard.



The behaviour of this feature can be configured in **Window > Preferences**. In *Figure 4* shows the configuration page of the content wizard.
CCE workbench – Preferences window.

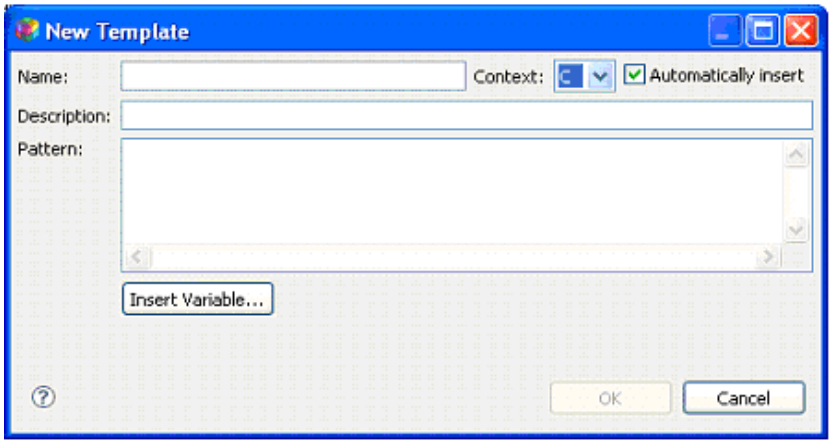


The search range can be restricted to only the edited file and to the files included therein (**Search current file and included files**), or alternatively a search can be in the whole project (**Search current project**). Automatic model insertion is allowed, as long as it is the only one at the options list (**Insert single proposals automatically**). The user may also request that the

suggestions list is presented in alphabetical order (**Present proposals in alphabetical order**). Another aspect that can be configured is related to the time (in milliseconds) that the content wizard delays to suggest a list (**delay**), or the duration of the validity of the suggestion (**Content Assist Parsing timeout**).

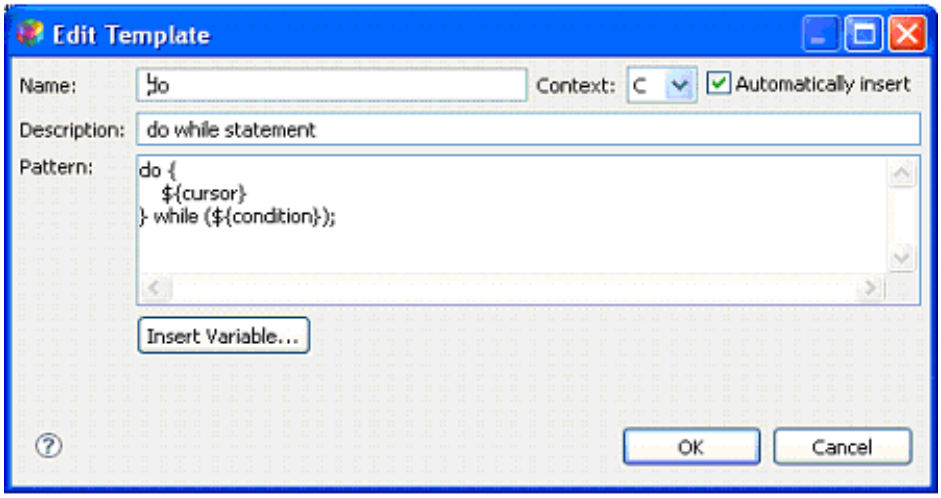
In addition to the sequence of *Ctrl + Space* keys, the content wizard can also be set automatically when the following characters are typed: ".", "->" or "::".

CCE is already provided with a set of models. However, it is possible to create new models by opening the models editor. Expand the C/C++ perspective in **Window > Preferences**, and select **Editor > Templates**. The option **New** must be selected to create a new model (see *Figure 5*).
CCE workbench – New template window.



A name must be given for the new model. The context in which the model is valid should be selected. In the **Description** field a brief description of the model can be added. The model itself is described in the **Pattern** field. To insert a variable, use the **Insert Variable** option.

One way to learn how to create models, or even how to customize existing models, can be achieved using the model editing feature (see *Figure 6*). To access this feature, the **Editor > Templates** option must be chosen, and is visible after expanding the C/C++ perspective in **Window > Preferences**.
CCE workbench – Edit template window.



The procedures to check on this page are identical to those described earlier for building new models.

The CCE supports the following intrinsic functions for the MSP430 family devices:

- void __no_operation(void);

- void __enable_interrupt(void);

- void __disable_interrupt(void);

- unsigned short __get_interrupt_state(void);

- void __set_interrupt_state(void);

- void __op_code(unsigned short);

- unsigned short __swap_bytes(unsigned short);

- void __bic_SR_register(unsigned short);

- void __bis_SR_register(unsigned short);

- unsigned short __get_SR_register(void);

- void __bic_SR_register_on_exit(unsigned short);

- void __bis_SR_register_on_exit(unsigned short);

- unsigned short __get_SR_register_on_exit(void);

- void __set_SP_register(unsigned short);

- unsigned short __get_SP_register(void);

- unsigned short __bcd_add_short(unsigned short, unsigned short);

- unsigned long __bcd_add_long(unsigned long, unsigned long);

- void __data20_write_char(unsigned long, unsigned char);

- void __data20_write_short(unsigned long, unsigned short);

- void __data20_write_long(unsigned long, unsigned long);

- unsigned char __data20_read_char(unsigned long);

- unsigned short __data20_read_short(unsigned long);

- unsigned long __data20_read_long(unsigned long);

Request the MSP430 Teaching ROM Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

File history

File history

Another of the features included in CCE allows comparisons between two files or previous versions of it, using file history stored during the work sessions. This feature allows searching and integrating the different versions between files.

The file to compare with the local history must be selected in one of the navigation views. In the context menu (select the file, mouse right button click), choose the **Compare With > Local history** option. In response to this selection, the **Compare With Local History** window is opened. A previous state presented in the **Local History** list can be chosen. The text comparison editor will then be open. The navigation between changes is made through the buttons **Select Next Change** and **Select Previous Change**.

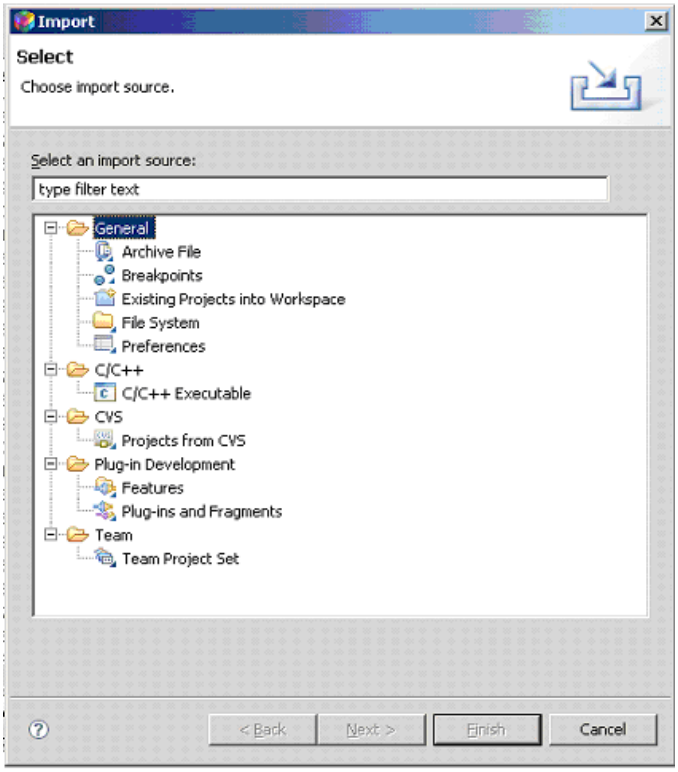
It is possible to recover a resource that has been cleared of the workspace. The procedure is as follows: the project to which is to be restored to a previous state must be chosen in the navigation view. In the context menu, the **Restore from Local History...** option should be chosen. The **Restore From Local History** window will open on the right hand side of the screen. It will display all the files that were previously part of the project. The last file version, or any of those previous, can be fully recovered by choosing it in the **Local History** list. The restore will be done after clicking **Restore**.

The file history feature can be configured according to Project needs. In the preferences page **General > Workspace > Local History**, it is possible to establish the number of days that a particular file history should remain available and the maximum number of entries per file. If the defined value is exceeded, the oldest changes are removed in order to provide memory space for the latest. The maximum size available to store the file history can also be defined. If its size is exceeded, the file history ceases to be performed.

Request the MSP430 Teaching ROM Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

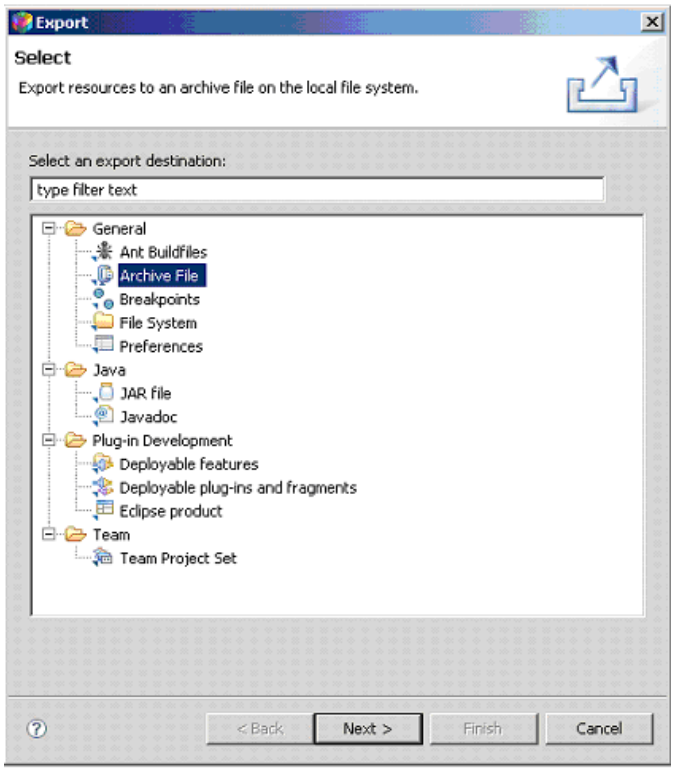
Import and Export functionality

CCE has the capability to import and export different types of information. In the context menu of the view **C/C++ Projects** it is possible to activate the import process choosing the option **Import**. This process allows importing resources such as those listed in the following figure (*Figure 1*).
CCE workbench – Import options window.



Following the instructions given by the import wizard: **Archive File** (imports files stored in a compressed file); **Import Breakpoints** (imports a breakpoints scenario previously defined in another or in the same project); **Existing Project into Workspace** (imports a project into the actual workspace); **File System** (imports a file); **Preferences** (imports the CCE configuration preferences), etc.

When the **Export** option from the context menu is selected, the window with the export procedures is displayed, as shown in *Figure 2*.
CCE workbench – Export options window.



Similar to the import procedure, the resources belonging to a Project can be exported: **Archive File**; **Export Breakpoints**; **File System**, **Preferences**, etc…

Request the MSP430 Teaching ROM Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Project Configuration details

Project Configuration details

The project configuration defines a set of options to build it. The options defined at this level are applied to all the files of the project. CCE allows setting different options for building at different stages of the project.

Building a project is a process that generates new features starting from the existing ones, or updates them if they already exist. In the workspace, different builds for different types of projects, or for different stages of development can be invoked. The different build types are:

- An **Incremental build** uses a build held earlier. Thus, from a past build state, it applies the necessary changes to the resources that have been changed;

- A **Clean Build** ignores all previous builds as well as problems or errors that led to them. This type of build will transform all resources in accordance with the set of rules in the project build configuration.

The project builds can be done in two different ways. The behaviour configuration can be defined in **Window > Preferences > General > Workspace**:

- Automatic builds are always incremental and are always carried out throughout the workspace. Whenever there is resource alteration, it will initiate a build process. This option may be disabled in **Window > Preferences > General > Workspace**;

- A manual build is always triggered by the user. This type of project build option can be clean or incremental, and can be applied on a group of project files, or to the whole workspace.

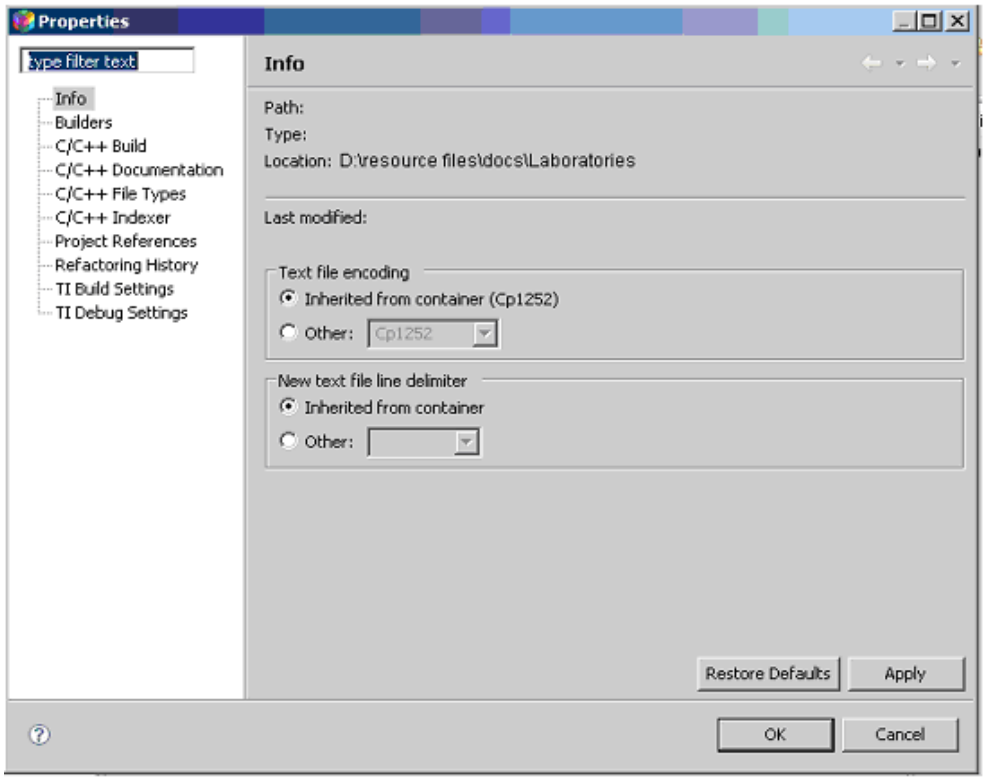
The order in which the build is processed is configurable. If the project contains mentions to another project, the CDT (C/C++ Development Tools) must first build the initial project. The order in which the build takes place

may be selected in **Window > Preferences > General > Workspace > Build Order**.

In order to bring the various parts of a project together, it is necessary to build the project using a configuration stored in a file. There are several build files available, giving different build alternatives, so the build file most appropriate to the stage of the project must be selected. The CDT can automatically generate build files whenever a **Managed Make C project** or **Managed Make C++ project** is created. Each project is therefore created with two default settings: Debug and Release. Other additional settings can be configured. Whenever a project is created or an existing project is opened, the first configuration in the list of alphabetically sorted items, is taken as active.

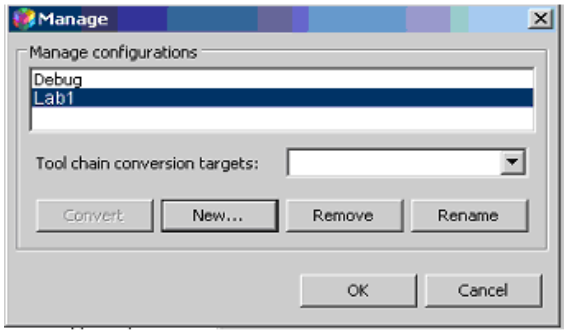
The project's compiler and linker definition options are complex. Therefore, it is recommended to carefully read the documentation related to the compiler and to the assembler/linker.

After the project's creation, it must be configured for the appropriate compiler, linker and debugging options. By selecting the option **Properties** from the context menu of the view **C/C++ Project**, the project's configuration window is displayed (*Figure 1*). The compiler, linker and debug options can be defined here.
CCE workbench – Configuration window.



The management of build configurations is found under the option **C/C++ Build**, accessed via the **Manage** button. Through it the management features can be accessed (see *Figure 2*).

CCE workbench – Manage window.



It is possible to create new build configurations, delete the existing ones or modify their names. The name of the modified configuration is selected in **configuration**.

The C/C++ compiler used by the project is controlled by the project's properties. To view the project properties in the dialog box that appears, the

page **C/C++ Build** allows control of the variety of configurations, including:

Build Options: specifies the options that affect all project files. This dialog page allows selection of the appropriate options, including those for compiling and linking. It is also possible to specify whether the compiler uses **Stop On Error** or **Keep Going On Error**. The second option allows the compiler to build all projects referenced, even if the current project contains errors. The build command specifies the make file to use.

The MSP430X devices allow data to be located anywhere in the 20-bit address space. By enabling this option, the compiler will use instructions that need a larger space for their storage. Hence, the memory space occupied by the final program will be greater. The option (- large_memory_model) is valid only when the project is intended as a MSP430X device defined by the compile option (- vmspx). The programs for MSP430X processors should be compiled with RTS libraries supplied for that purpose (rts430xl.lib and rts430xl_eh.lib).

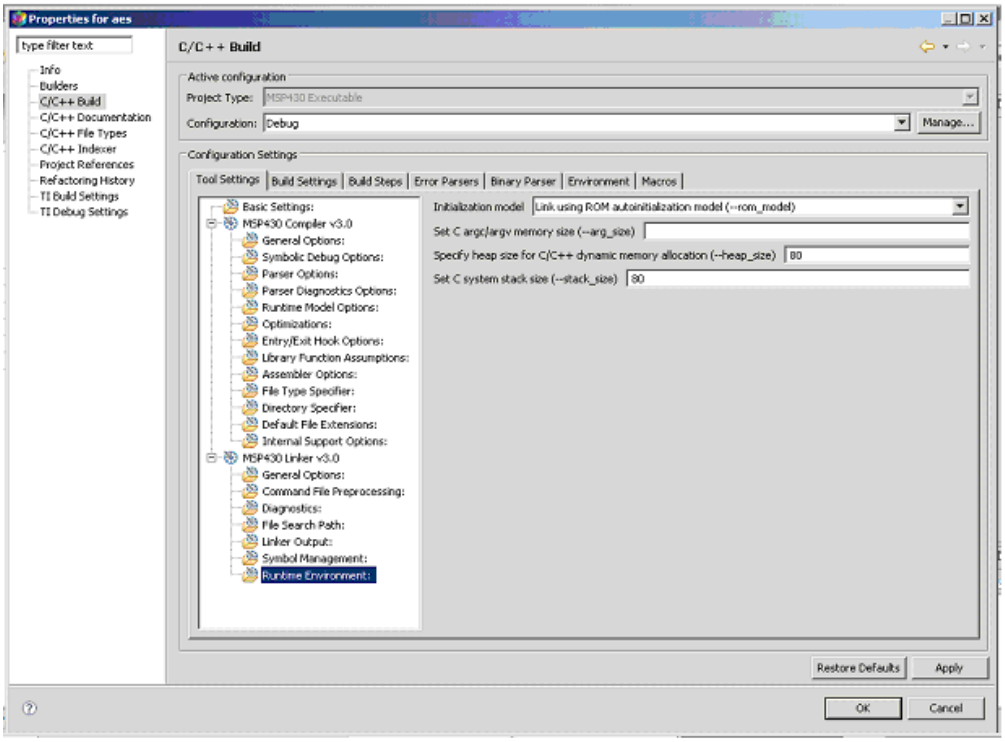
The compilation option (- silicon_version) selects the CPU version using the 4 least significant processor's identification digits. If this option is not used, the compiler will construct the default code for the device.

In the process of linking, if all references to the multiplication routines of integers are to be replaced by the routines versions that use the hardware multiplier option (- use_hw_mpy), the device multiplier's length must be specified. To use the 16-bit hardware multiplier, present in most devices, choose the option 16. For devices belonging to the F4xx family, which has a 32-bit multiplier module, chose the option 32. Finally, for the new 5xx family, which also has a 32-bit multiplier, use the F5 option. The default option is 16-bit hardware multiplier module.

The model used for the initialization of static variables in the C program can be specified as: None, Link using ROM autoinitialization model (- rom_model), or link using RAM autoinitialization model (- ram_model). The C/C++ compiler produces tables of data for automatic initialization of global variables. These tables are placed in the section identified by .cinit.

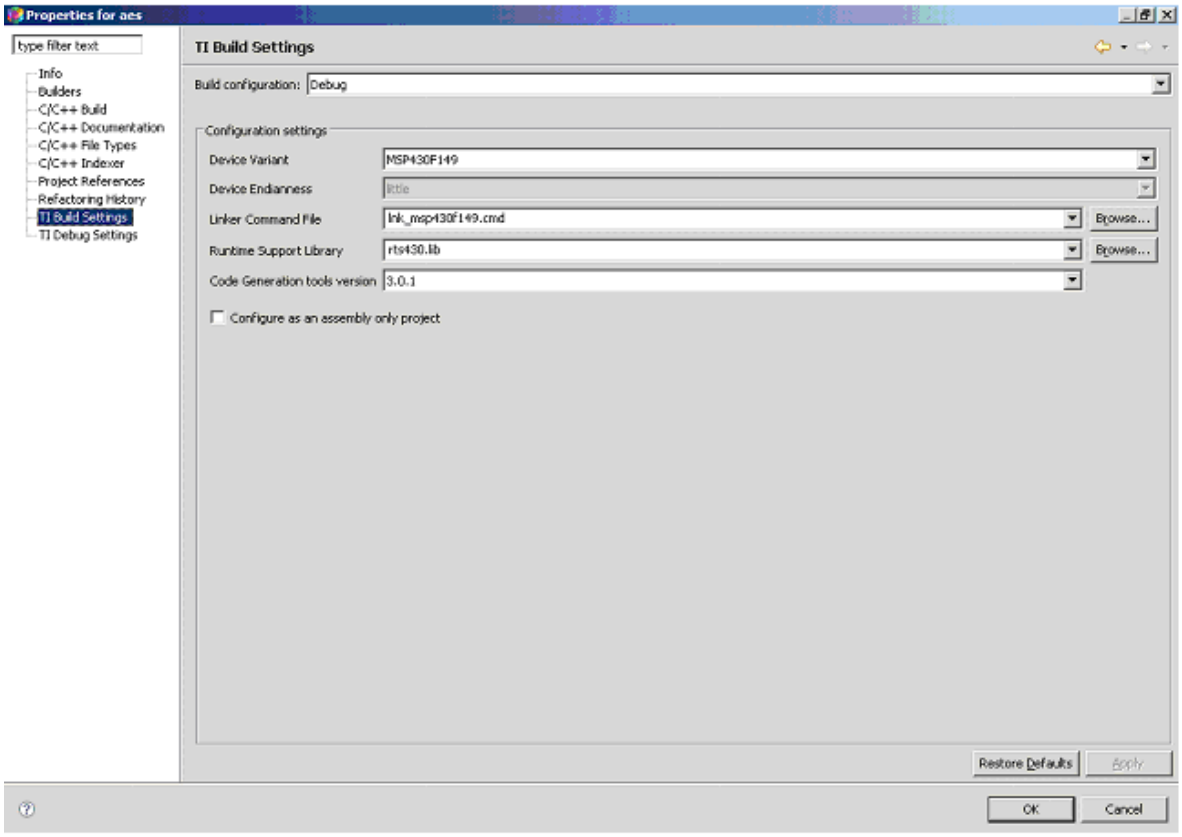
The memory space reserved for the passing of arguments by the C routines is defined in (--arg_size). The space reserved for the dynamic allocation of memory by the program is defined in the option (--heap_size). The system stack size used by the program is set by the option (--stack_size). See *Figure 3*.

CCE workbench – Memory space configuration.



The device for which the project is intended is configured in the **Properties> TI Building Setting**. The window is in every way identical to that presented in the project's creation (*Figure 4*).

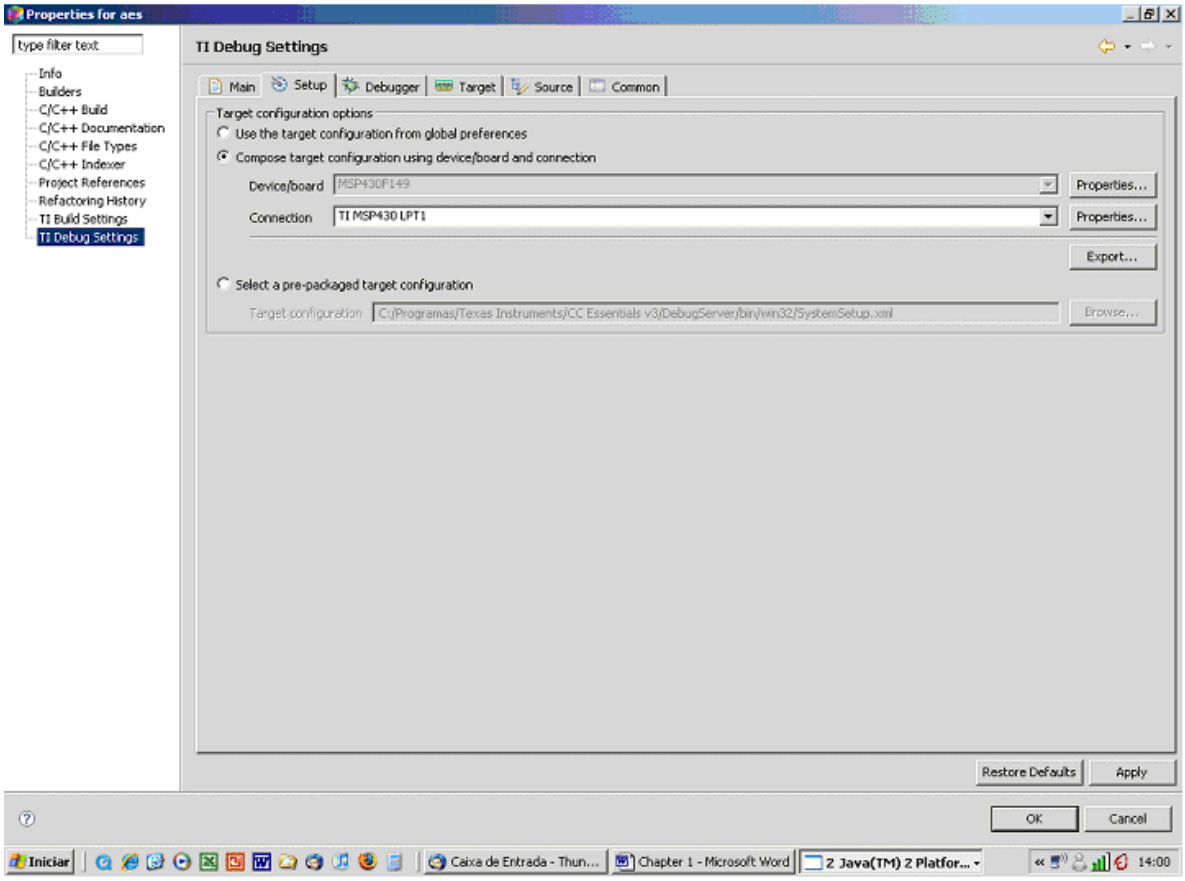
CCE workbench – Device configuration.



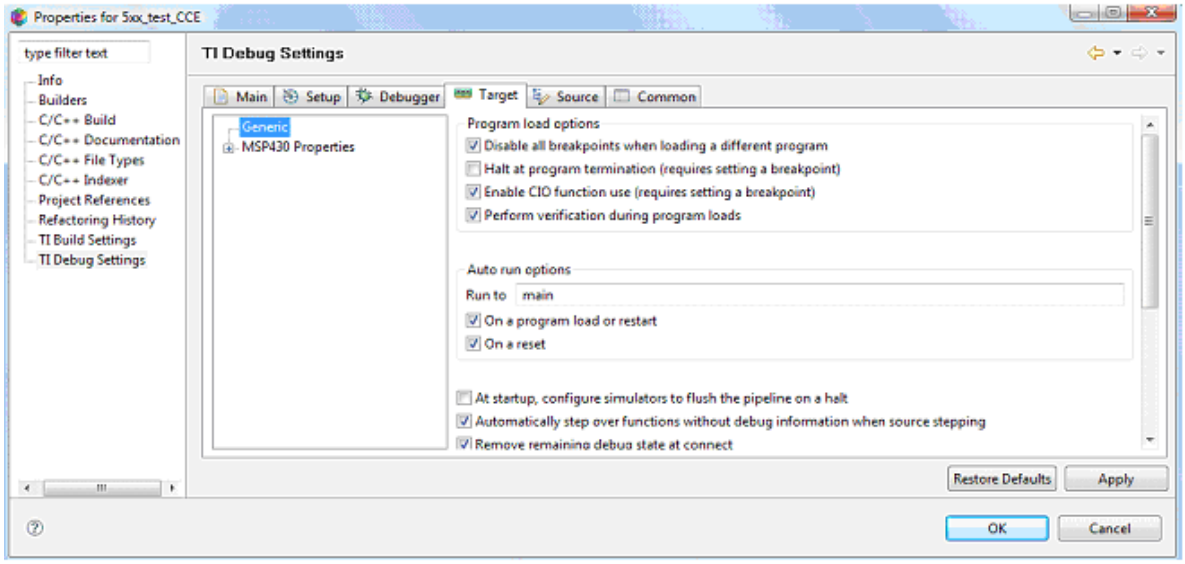
The project debugging is carried out as specified in the window **TI Debug Settings**. With the **Setup** tab, using the option **connection**, the method of connecting to the device is established, either parallel port or USB port. The **Debugger** tab can be used to specify whether to load the all application (**Load program**) or just load the project's symbols (**Load symbols only**). These options can be used to choose between loading the entire program, or just the symbols. This last option is valid when the development environment cannot load the software, such as in the case of the software runs in ROM.

Using the **Target** tab, it is possible to define various aspects related to the device. Thus, it is possible to enable the use of IO functions in **Enable CIO functions** use, or establish the starting point for the code execution when a reset occurs or a program is loaded. In the MSP430 properties, it is possible to specify the supply voltage and the types of breakpoints: software or hardware. The memory storage process can also be defined using this tab (*Figures 5 to 7*).

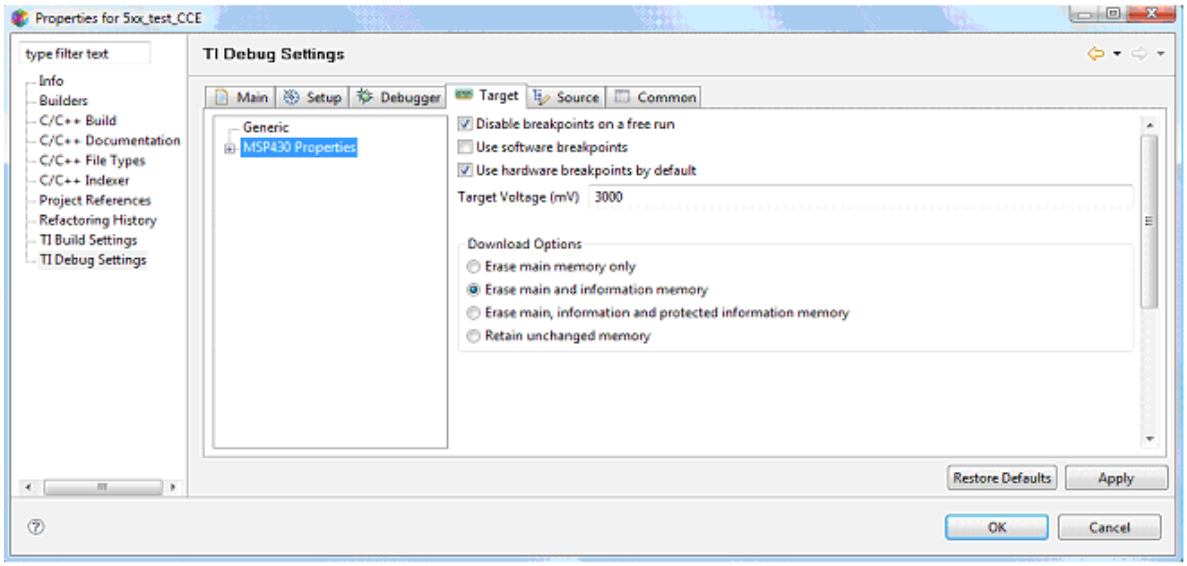
CCE workbench – Device options configuration.



CCE workbench – Device options configuration: TI Debug Settings – Target: Generic.



CCE workbench – Device options configuration: TI Debug Settings – Target: MSP430 properties.



The first time that the project is built, the **Project > Build All** option must be selected. The project build status can be examined in the **Console** window. If there is a problem, the **Problems** window will list them all. After a successful build of the project, the output file can be automatically loaded into the device.

Alternatively, a project can be built at the beginning of the debug session. The option **Debug Active Project** will recompile the project and launch the debugger, using the device information defined in the project options.

Note that an attempt to update the **firmware** can occur when the debugger is used for the first time, after a software release has been installed or a new USB interface is used.

Finally, the active perspective must be switched to the **Debug** perspective. This operation can be carried out with the perspective selection buttons located on upper right corner of the workspace window, or alternatively, by selecting **Window > Open Perspective > Debug**.

When the project is debugged, the errors are identified on the right hand side of the editor as red marks while the problems are identified as white marks. A mark is added on the left hand side of the editor to the lines that contain an error. When this mark is selected, the compiler provides information about the error.

When the project is made (make option), the resources used can be accessed on **Properties > C/C++ Build > MSP430 Linker V3.0 > Linker Output** in the option **Produce list of input and output sections**.

Request the MSP430 Teaching ROM Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Introduction to Debug with CCE

Introduction to Debug with CCE

TI’s debugger generates an output file as a result of building the project. To debug a project after a build, it is only necessary to perform the following steps:

1. Select the project as active, or click **Run > Debug Active Project**. The debug perspective is open and it is possible to debug the code;

2. This resets and suspends the execution of code on the device. Running this command, the content of all status registers is modified to the power-up state defaults in accordance with the device specifications. The reset command is enabled by **> Reset CPU**;

3. To start a program execution, once loaded into the device's memory, select the option **Run > Run** (F8). Program execution will take place until the program finds a breakpoint;

4. The program execution may be suspended at any time by using the command **Run > Halt**;

5. To re-start the execution of the application use the command **Run > Restart**. This action does not modify the execution stage of the device. It only restores the PC to the application’s starting point loaded into memory;

6. The **Set PC to Cursor** feature moves the execution of the application to a particular point in program memory. The execution of this command only modifies the contents of the PC register. No instruction will be executed in order to reach this point. The command can be found in the context menu of the C/C++ perspective in **Set PC to Cursor**;

7. There are several different ways to run the code until a specific point:

- Use a breakpoint to specify that when this point is reached the program execution must be halted;

- Use the command Run > Run to Line, available in the context menu of C/C++ perspective, to run the code until the specified location;

8. A special case is to run the code until the main function is reached. This feature enables a temporary breakpoint at the beginning of the main routine and starts the execution of the application. The breakpoint is removed and execution is suspended once the location is reached. This command provides a convenient method of starting C applications.

9. The stepping commands execute each instruction step-by-step. When a function is called, it is possible to move the execution to the function (step into) or perform the function and pass to the following instruction (step over). Once inside a function, the user can continue to execute each instruction individually, or run the rest of the function code until it ends (step out);

10. The execution of the next instruction is performed through **Run > Step Into** (F5). The next instruction is executed when this command is used. If the next statement is a function call, the debugger passes the execution to the first instruction of the function, and suspends execution at this point;

11. When the execution is on top of a function call, the step over operation can be enabled by selecting the **Run > Step Over** (F6). The debugger then performs the function and then suspends execution when it returns. If it finds a breakpoint somewhere in the function, the execution may be suspended at this point. If the **Step Over** is executed on an instruction that is not a function call, the debugger response will be the same as **Step Into** command;

12. If the application is being executed inside a function in response to a function call, it is possible to force the return of this function through the command **Run > Step Return** (F7). The debugger will execute the rest of the function code and return the calling point. The execution will be suspended at this point;

13. The command **Terminate** allows finishing of the application’s debugging.

EEM - Enhanced Emulation Module).

All the MSP430 family of devices have an advanced code debugging module (EEM - Enhanced Emulation Module). This module allows CCE to monitor the device's operation in a non-intrusive way, and without using any resources. Thus, it facilitates the development of the application through the verification of its operation. Depending on the device, the EEM module implementations differ. Generally, the following features are present:

- Up to 8 hardware breakpoints;

- Operates in all range of frequencies and clock sources;

- Ability to set more complex breakpoints through association of triggers;

- Suspend the execution of the application on the occurrence of a program or data bus access;

- Access protection to protected data and program memory areas;

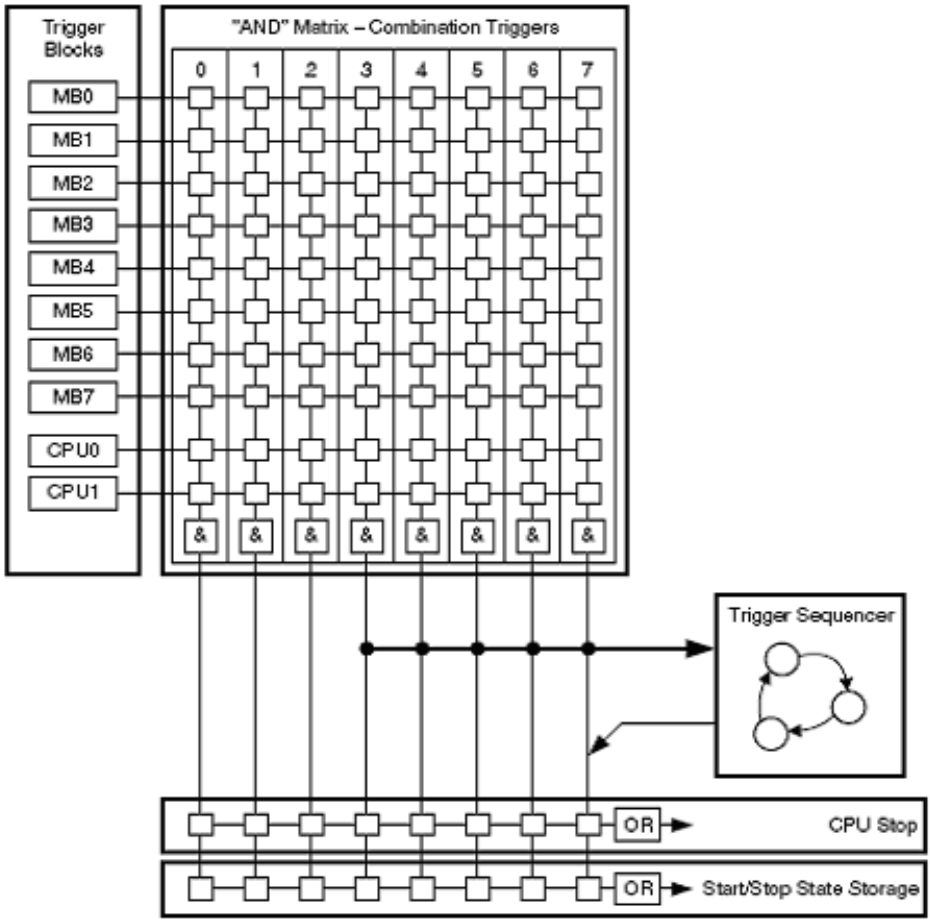
- All timers and counters can be inhibited (depending on the device);

- Inhibits PWM signals generation on the occurrence of the application's suspension;

- Allows real-time execution of the applications in the modes: single step, step into; run to cursor; step over;

- Supports all low power modes.

The *Figure 1* represents a simplified block diagram of one of the most complete implementations of EEM module.
CCE workbench – EEM module block diagram.



Events within the device can generate triggers. These triggers can be classified as the event that causes them to:

- Access to addressing and data buses;

- Access to CPU registers.

Depending on the device, it is possible to associate two or more of these triggers, in order to build complex event detectors that help the detection of incorrect operation of applications. Generally, a trigger can be used to control the following functional blocks of the EEM: breakpoints, trace, and sequencer. The activation of a trigger is conditioned to an access to the data and program busses or access to CPU registers.

A breakpoint is set through one or more triggers. Through these it is possible to establish the following types of breakpoints:

- Address breakpoint;

- Data breakpoint;

- Register Breakpoint;

- Mask Register;

- Range breakpoint.

A simple breakpoint is defined using a trigger associated with an instruction read operation by the CPU. It is necessary to specify the instruction address where the trigger should occur.

By combining two triggers, it is possible to establish a Data Breakpoint. While one of the triggers is used to detect the occurrence of a particular address on the address bus, the other is used to detect the occurrence of a read or write operation at that address. It is possible to force the suspension of the execution of the application to only occur when there is a match between the value written or read and the one specified.

When the application is written in assembly language, it is sometimes necessary to analyse the accesses to some of the microcontroller's registers. A Register Break Point uses a trigger to detect the access to a register. A Mask Register should be used when the register is composed of several fields, since it can apply a mask and test specific bits only.

An application in certain operating conditions may occasionally try to access to invalid or protected memory regions. Using a range breakpoint, it is possible to detect the occurrence of these events. It is thus possible to suspend the execution of the application on the occurrence of:

- Write to flash;

- Invalid access to memory;

- Access to an instruction in invalid program space;

- Access to data in invalid data space.

The hardware breakpoint properties are established through different fields. The action to make when all triggers are true can be defined in the Action option of the Hardware Configuration field. One of the following options can be chosen:

- Halt;

- Trigger storage;

- Halt and trigger storage.

In the trigger field, specify through various options, the check condition for a true trigger. The trigger can be:

- Memory Address bus;

- Memory Data bus;

- Register Write.

Depending on the type of trigger chosen, the options to specify may be

Memory Address Bus

Location: Address of the program code line or data memory address (e.g.: &a);

Mask: the information introduced in this field is used in a logic AND operation with the contents;

Operator: Logic operation with the data (==, <=, >=, !=);

Access: Memory access type:

- Instruction fetch;

- Instruction fetch and hold trigger;

- No instruction fetch;

- Don’t care;

- No Instruction fetch and read;

- No instruction fetch and write;

- Read;

- Write;

- No instruction fetch and no DMA access;

- DMA access (read or write);

- No DMA access;

- Write and no DMA access;

- No instruction fetch and read and no DMA access;

- Read and no DMA access;

- Read and DMA access;

- Write and DMA access.

Memory Data Bus

Value: A mask and compare will be applied to the data on the bus and to value added here, to determine if the trigger is true;

Mask: The information introduced in this field is used in a logic AND operation with the contents;

Operator: Logic operation with the data (==, <=, >=, !=);

Access: Memory access type (on Memory Address Bus).

Miscellaneous

Group: Group to which the breakpoint belongs;

Name: Name assigned to the breakpoint.

There is a predefined breakpoint that can be set to

- **Break in program range**: Generates a suspension of the execution of the application in a range of program memory addresses. It uses two triggers that define the range of addresses;

- **Break in DMA transfer**: Generates the suspension of the execution of the application, whenever a DMA read or write operation at the specified program address occurs. This breakpoint is implemented using only one trigger;

- **Break in DMA transfer range**: Generates the suspension of the execution of the application, whenever a DMA read or write operation at the specified address range occurs. This breakpoint is implemented using two triggers;

- **Break in stack overflow**: Generates the suspension of the execution of the application whenever the SP register value assumes a lower value than the specified one. This breakpoint is implemented using only one trigger;

- **Breakpoint**: Generates the suspension of the execution of the application whenever the memory bus address takes the value specified. This breakpoint is implemented using only one trigger;

- **Hardware breakpoint**: Generates the suspension of the execution of the application whenever the memory bus address takes the value specified. This breakpoint is implemented using only one trigger;

- **Watch on data address range**: Generates a suspension of the execution of the application whenever the specified data memory addresses range is accessed. It uses two triggers to define the range of addresses;

- **Watchpoint**: Generates the suspension of the execution of the application whenever a specified data memory address is accessed. It uses a trigger to generate the watchpoint;

- **Watchpoint with data**: Generates the suspension of the execution of the application whenever a specified data memory address is accessed and the value of the address is equal to specified value. Two triggers are used to implement this watch.

Code Execution Verification

In order to verify the code execution, it is necessary to use support tools to complete this task. CCE provides a set of features with this aim.

A breakpoint suspends the execution of the application in order to check the status of the system. The activation, deactivation and configuration of these breakpoints are possible through CCE.

There are two types of breakpoints: software and hardware. While the first type of breakpoint is implemented through the insertion of code in the application, in a way invisible to the user, the second type is implemented internally by the device's hardware. Although the software breakpoints are not limited, the hardware breakpoints, depending on the device, have a limit of 2 to 4 breakpoints.

The application debugging process often requires access to the actual values of the variables. The **Variable** view allows the user to monitor the application's local and global variables. In this view, the CCE automatically

displays the name and contents of the local variables of the function that is being executed. It is also possible to add the name of other local variables or global variables to be monitored in the debugging process.

The values of the local variables can be modified. The values of the variables that have been changed during the last instruction execution are displayed in red. However, the variable names cannot be modified. It is allowed to change the representation format of the variable: Natural, decimal or hexadecimal. The variables that contain more than one element, such as arrays, structures, or pointers are presented with a (+) sign immediately after the name. This signal means that the variable has elements that can be seen through the expansion of the (+) sign, passing this signal (-), which allows the structure to be collapsed.

The local variables cannot be added or removed from the Variables view. However, global variables can be added or removed. The local variables can be disabled in order to freeze their value as the program is executed.

The **Expressions** view accepts the entry of expressions to evaluate them as the program is executed. These expressions are written in syntax similar to that used by the C programming language.

The commands accessible through the context menu can

- Specify the number of elements of the array to be displayed in the Expression view: The command **Display as Array** can be used to display the elements of any pointer or array. The command **Remove Array Expansion** is used to return an expanded variable back to its original state;

- Change value: Changes the content of the variable;

- Cast to type: Performs a promotion (cast) for a different type of variable;

- Restore Original Type: Restores the expression for the original data type.

The Memory window of the Debug perspective

The **Memory** window of the Debug perspective allows the user to monitor and modify the device's memory. The memory is provided with a list of **Memory Monitors**. Each monitor represents a section of memory specified by its named location base address. Each memory monitor may be represented in different data formats (memory renderings). The debugger allows four different types of rendering:

- Hex (default);

- Ascii;

- Integer signed;

- Unsigned integer.

The Memory view has two panels:

- Memory Monitors;

- Memory Renderings.

The first panel displays the memory monitors list added to the debug session. The second panel is controlled by selection in the first one and consists of tabs that display the rendering. This panel can be configured to display both renderings.

Request the MSP430 Teaching ROM Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Laboratory GPIO: Lab1 - Blinking the LED
Using the MSP-EXP430FG4618 Development Tool and the eZ430 kits blink the LED1.

Laboratory GPIO: Lab1 - Blinking the LED

Introduction

The hands-on laboratory consists of configuring the I/O ports, setting up the input lines to read push buttons and the output lines to feed LEDs. The following exercises have been developed for the three hardware development tools.

The first to be discussed is the MSP-EXP430FG4618 Experimenter's board. Modifications are later made to suit the other development boards. The main differences between the boards are related to the specific ports in which the buttons and LED are (or can be) connected. For the development of this laboratory, Code Composer Essentials v3 has been used.

Procedure

By analysis of the schematics, determine which I/O port pin is connected to the LED on the board:

- Consult the MSP430FG4618/F2013 Experimenter's Board User's Guide slau213a.pdf

- LED1 is connected to Port 2.2

- Consult the eZ430-F2013 Development Tool User's Guide slau176b.pdf

- LED1 is connected to Port 1.1

- Consult the eZ430-RF2500 Development Tool User's Guide slau227c.pdf

- LED is connected to Port 1.0

Include the standard register and bit definitions for the TI MSP430 microcontroller device (example for the MSP430FG18/MSP430F2013 Experimenter's board):

```
#include <msp430xG46x.h>
```

Define the main routine:

```
void main (void){
```

The watchdog timer must be prevented from generating a PUC. Write 0x5A to the eight MSBs of the Watchdog timer control register, WDTCTL:

```
WDTCTL = WDTHOLD | WDTPW;
```

Port control registers:

- Set the LED port pin as an output;

P2DIR: Port 2.2 is set as an output:

```
P2DIR |= 0x04; // to force the pin setting. It is uses an OR operation ( | ) with P2DIR and 0x04
```

Use an infinite loop to modify the state of the port;

Use a software delay loop to generate the pause interval. (a long software delay loop is used here for simplicity - in real applications, a timer would be used)

- Because no clock is defined, the device will use the 32.768 kHz watch crystal. In order for a rate of one blinking LED state transition each second, the software delay loop should count to approximately 30000 {30000/32768 = +/- 1 sec};

volatile unsigned int i;

```
while(1){ //Infinite loop i=30000; //Delay do (i--); while (i !=0);
```

- Port control registers inside the loop:

P2OUT: To switch the port state between low and high state during program execution:

```
P2OUT ^= 0x04}}; // It uses an XOR operation ( ^ ) between P2OUT and 0x04:
```

- The programming code for the other hardware kits follows the same sequence as given above, requiring only configuration the port.

This example and many others are available on the MSP430 Teaching ROM.

Request this ROM, and our other Teaching Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Laboratory GPIO: Lab2 - Blinking the LED half the speed
Using the MSP430-EXP430FG4618 Development Tool and the eZ430 kits blink the LED1 half the speed.

Laboratory GPIO: Lab2 - Blinking the LED half the speed

Introduction

The hands-on laboratory consists of configuring the I/O ports, setting up the input lines to read push buttons and the output lines to feed LEDs. The following exercises have been developed for the three hardware development tools.

The first to be discussed is the MSP-EXP430FG4618 Experimenter's board. Modifications are later made to suit the other development boards. The main differences between the boards are related to the specific ports in which the buttons and LED are (or can be) connected. For the development of this laboratory, Code Composer Essentials v3 has been used.

Procedure

Using the *Lab1: Blinking the LED* example, independently of the hardware development tool, reduce the value of the software delay to half its previous value.

```
#include "msp430xG46x.h" void main (void){
volatile unsigned int i; WDTCTL = WDTPW | WDTHOLD;
// Stop Watchdog Timer P2DIR |= 0x04; // Configure
P2.2 as Output while(1){ // Infinite loop i=30000;
// Delay do (i--); while (i !=0); P2OUT ^= 0x04;
// Toggle Port P2.2 using an exclusive-OR } }
```

- In order for a rate of two blinking LED state transition each second, the software delay loop should count to approximately 15000 {15000 / 32768 = 0.5 sec};

```
i=15000; //Delay
```

This example and many others are available on the MSP430 Teaching ROM.

Request this ROM, and our other Teaching Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Laboratory GPIO: Lab3 - Toggle the LED state by pressing the push button Use the S1 button in the MSP430-EXP430FG4618 development tool and in the eZ430-RF2500 kit to toggle the state of LED1. You must use interrupts to catch the button press and you should ensure that the MSP430 is in Lower Power Mode 3 when is not in use.

Laboratory GPIO: Lab3 - Toggle the LED state by pressing the push button

Introduction

The hands-on laboratory consists of configuring the I/O ports, setting up the input lines to read push buttons and the output lines to feed LEDs. The following exercises have been developed for the three hardware development tools.

The first to be discussed is the MSP-EXP430FG4618 Experimenter’s board. Modifications are later made to suit the other development boards. The main differences between the boards are related to the specific ports in which the buttons and LED are (or can be) connected. For the development of this laboratory, Code Composer Essentials v3 has been used.

Procedure

By analysis of the schematics, determine to which port pin the push button is connected:

- Consult the MSP430FG4618/F2013 Experimenter’s Board User's Guide <slau213a.pdf>:

- Button S1 is connected to Port 1.0;

- Consult the eZ430-RF2500 Development Tool User's Guide <slau227a.pdf>:

- Button S1 is connected to Port 1.2;

- The eZ430-RF2500 uses a device in 2xx family, so you need to additionally configure the button as pull-up or pull-down, in the P1REN register.

Ports control registers:

- Set push button pin port as an input

- P1DIR: Port 1.0 is set as an input:

```
P1DIR &= ~0x01 // to force the pin setting to 0. It is uses an AND operation ( & ) between P1DIR and 0xFE
```

- Enable interrupts to this pin port;

- P1IE: Enable interrupt to port 1.0:

```
P1IE |= 0x01; // Interrupt Enable in P1.0
```

- PIIES: Call the port interrupt on a high-to-low transition:

```
P1IES |= 0x01; // P1.0 Interrupt flag high-to-low transition
```

- Configure the watchdog timer to prevent a PUC during the program execution;

```
WDTCTL = WDTPW | WDTHOLD; //Stop Watchdog Timer
```

- Enable Global Interrupts and configure low power mode 3;

```
_BIS_SR (LPM3_bits + GIE); //Low Power Mode with interrupts enabled
```

- Create a interrupt service routine, that includes:

- Toggle LED1 pin port;

- Delay for button debounce;

- Clear interrupt flag.

```
#pragma vector=PORT1_VECTOR __interrupt void
Port_1 (void) { volatile unsigned int i; P2OUT ^=
0x04; // Toggle Port P2.2 i=1500; // Delay, button
debounce do (i--); while (i !=0); while (! (P1IN &
0x01)); // Wait for the release of the button
i=1500; // Delay, button debounce do (i--); while
(i !=0); P1IFG & = ~0x01; // Clean P1.0 Interrupt
Flag }
```

This example and many others are available on the MSP430 Teaching ROM.

Request this ROM, and our other Teaching Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Laboratory GPIO: Lab4 - Enable/disable LED blinking by push button press
Use the S1 button in the development tool to start and stop the led blinking. It is not necessary to use Lower Power Modes in this exercise.

Laboratory GPIO: Lab4 - Enable/disable LED blinking by push button press

Introduction

The hands-on laboratory consists of configuring the I/O ports, setting up the input lines to read push buttons and the output lines to feed LEDs. The following exercises have been developed for the three hardware development tools.

The first to be discussed is the MSP-EXP430FG4618 Experimenter's board. Modifications are later made to suit the other development boards. The main differences between the boards are related to the specific ports in which the buttons and LED are (or can be) connected. For the development of this laboratory, Code Composer Essentials v3 has been used.

Procedure

Detect of the button is pressed:

```
if (!(P1IN & 0x01))
```

Include a control flow program variable that detects if the LED is blinking or not, when the button is pressed:

- Define a variable that indicates whether the LED is blinking;

```
unsigned char blink_status=1;
```

- Set the program flow depending on the state of the variable.

```
while(1){                          // Infinite loop
 if (blink_status == 1) {
  P2OUT ^= 0x04;                   // Toggle Port P2.2
  i=15000;                         // Delay
  do (i--);
   while (i !=0);
 }
 if (!(P1IN & 0x01)) {     // Detect S1 pressed
  i=1500;                          // Delay, button
debounce
  do (i--);
  while (i !=0);
   while (!(P1IN & 0x01)); // Wait for the release
of the button
    i=1500;                        // Delay, button
debounce
    do (i--);
     while (i !=0);
      if (blink_status ==1){   // If led is
blinking, stop it
       P2OUT&= ~ 0x04;     // Turn Led off
       blink_status=0;
      }else
       blink_status=1;
```

This example and many others are available on the MSP430 Teaching ROM.

Request this ROM, and our other Teaching Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Laboratory Timers: Lab1 - Memory clock with Basic Timer1
Using the MSP-EXP430FG4618 Development Tool and the MSP430FG4618 device implement a Memory clock with Basic Timer1.

Laboratory Timers: Lab1 - Memory clock with Basic Timer1

Introduction

Correct system timing is a fundamental requirement for the proper operation of a real-time application. The timing definition can dictate how the data information processed during the execution of the application program. The clock implementations vary between devices in the MSP430 family. Each device provides different clock sources, controls and uses. This chapter discusses the clock controls included in the platforms used. The MSP430 4xx family has two general-purpose 16-bit or 8-bit counters and event timers, named Timer_A, Timer_B, and a Basic Timer. The Basic Timer module is only implemented in ‘4xx devices. The 2xx device family also has Timer_A and Timer_B, but the clock signals are provided by the basic clock module+. The timers may receive an internal or external clock. Timer_A and Timer_B also include multiple independent capture and compare blocks, with interrupt capabilities.

Overview

This laboratory implements a memory clock using the features provided by Timer1. The clock is updated once every second by the Basic Timer1 interrupt service routine (ISR). This procedure also performs switching of LED1. In order to evaluate the execution time of the routine, LED2 is kept active during the execution of the ISR. When the ISR has completed, the device goes into low power mode, until the new interrupt wakes it up.

Resources

This application (Lab1_Timers.c) sets Basic Timer1 to generate an interrupt once every second. The interrupt service routine generated by this peripheral is required to update the clock stored in memory. Moreover, it must refresh the content of the clock displayed on the LCD.

Thus, the system resources used by this application are:

- Basic Timer1;

- I/O ports;

- LCD;

- Interrupts;

- Low power modes.

The default configuration of the FLL+ is used, so, all the clock signals required for the operation of the components of the device assume their default values.

Software application organization

The first task is to disable the Watchdog Timer. It should be stated that this feature, when used correctly, makes the application more robust.

The resources needed for the LCD are all configured. This code is given, since its operation will be analysed in a later laboratory. Once the LCD configured, it is cleared by the execution of the routine `LCD_all_off()`.

The memory clock consists of setting three global variables: hour, min, and sec, all of the type unsigned char, used to store the hours, minutes and seconds values elapsed respectively since the beginning of the execution of the application. These variables are initialized with zero values.

The LCD is refreshed at startup to show the initial clock value.

LED1 is used as an indicator of Basic Timer1 ISR execution. The execution time can be determined through it. In addition, LED2 state switches whenever the Basic Timer1 ISR is executed.

The Basic Timer1 is set to generate an interrupt once every second.

The routine `main()` ends with the interrupts global activation and puts the device in low power mode, awaiting the next interrupt.

Basic Timer1 ISR begins by activating LED2, indicating the beginning of the routine execution and then switches the state of LED1. The counters are updated in cascade and their contents updated on the LCD, through routines `LCD_sec()`, `LCD_min()` and `LCD_hour()`. The routine ends with switching the state of the clock separation points. Finally, LED2 is turned off.

System configuration

Watchdog Timer

The Watchdog Timer is disabled with the objective of reducing energy consumption, but giving up the protection afforded by it. This peripheral is configured by the WDTCTL register. Its access is protected by a password. The value to disable it:

```
WDTCTL = WDTPW | WDTHOLD; // Stop WDT
```

FLL+ configuration

A 32.768 kHz crystal is applied to the oscillator LFXT1. Since it is possible to select the internal capacitors using software, the value to write to the FLL_CTL0 configuration register to select the 8 pF capacitors is:

```
FLL_CTL0 |= XCAP18PF; // Set load cap for 32k xtal
```

Taking into consideration the change mentioned earlier to the FLL+ module, what are the frequencies of each of the clock signals?

ACLK = _____;

MCLK = _____;

SMCLK = _____;

LED ports configuration

LED1 and LED2 are connected to ports P2.2 and P2.1 respectively. How should they be configured so that just the bits related to these ports have digital output functions?

```
P2DIR |= 0x06; // P2.2 and P2.1 as output
```

How should the P2OUT register be configured so that the application starts with LED1 on and LED2 off?

```
P2OUT |= 0x04; // LED1 on and LED2 off
```

Basic Timer1 configuration

Basic Timer1 should generate an interrupt once every second. It uses two counters in series, so that the input of the BTCNT2 counter is the output of the BTCNT1 counter divided by 256. The BTCNT1 counter input is the ACLK with a 32.768 kHz frequency. If the selected output of the BTCNT2 counter is divided by 128, what is the time period associated with the Basic Timer1 interrupt? _____

```
BTCTL = BTDIV | BT_fCLK2_DIV128; // (ACLK/256)/128
IE2 |= BTIE; // Enable Basic Timer1 interrupt
//*************************************************
********* // BasicTimer1 Interrupt Service Routine
```

```
//*************************************************
********* #pragma vector=BASICTIMER_VECTOR
__interrupt void basic_timer_ISR(void) { P2OUT
|=0x02; // LED1 turn on P2OUT ^=0x04; // LED2
toogle sec++; // increment seconds LCD_sec(); //
refresh seconds field in LCD if (sec == 60) // one
minute was pass { sec = 0; // reset seconds
counter min++; // increment minutes LCD_min(); //
refresh minutes field in LCD if (min == 60) // one
hour was pass { min = 0; // reset minutes counter
hour++; // increment hours LCD_min(); // refresh
hours field in LCD if (hour == 24)// one day was
pass { hour = 0; // reset hours counter } } } if
(sec & 0x01) // toogle clock dots { P3_DOT_ON;
P5_DOT_ON; } else { P3_DOT_OFF; P5_DOT_OFF; }
P2OUT &=~0x02; // LED1 turn off }
```

Low power modes

The task simply updates the counters periodically and refreshes the LCD contents. It is possible to configure the registers for an energy-efficient operation.

Which low power mode should be used? _____

Which system clocks are activated in the low power mode selected? _____

```
BIS_SR(LPM3_bits + GIE); // Enter LPM3 +
interrupts enabled
```

Analysis of operation

System clocks inspection

The MCLK, SMCLK and ACLK system clocks are available at ports P1.1, P1.4 and P1.5 respectively. These ports are located on the SW2, RESET_CC and VREG_EN lines, which are available on the H2 Header pins 2, 5 and 6. All these resources are available because the Chipcon RF module is not installed and SW2 is not used.

Using the Registers view, set bits 1, 4 and 5 of P1SEL and P1DIR registers, to choose the secondary function of these ports configured as outputs. By connecting an oscilloscope to those lines, it is possible to monitor the clock signals.

What are the values measured for each of the system clocks?

ACLK: _____

SMCLK: _____

MCLK: _____

ISR execution time

The Basic Timer1 ISR execution time is fundamental to energy conservation, in order to extend the life of the system battery. The routine execution time can be measured by connecting the oscilloscope to port P2.1, which controls LED2. This output is available on pin 2 of Header H4.

The execution time of this routine varies with the number of the counter updates and respective updates to the LCD. What are the times measured for each of these situations and what their frequencies?

Seconds update: _____ with a time period of _____

Seconds and minutes update: _____ with a time period of _____

LCD fields update: _____ with a time period of _____

If the developer chooses to update all the LCD fields at each interrupt, the time required is much greater than the solution presented. Efficient programming contributes to a reduction in the system power consumption.

Measurement of electrical current drawn

The power consumption was discussed in the previous point. The electrical power required by the system during operation is measured by replacing the jumper on the Header PWR1 by an ammeter, which indicates the electric current taken by device during operation.

What is the value read? _____

This example and many others are available on the MSP430 Teaching ROM.

Request this ROM, and our other Teaching Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Laboratory Timers: Lab2 - Real Time Clock with Basic Timer1
Using the MSP-EXP430FG4618 Development Tool and the MSP430FG4618 device implement a Real Time Clock with the RTC peripheral.

Laboratory Timers: Lab2 - Real Time Clock with Basic Timer1

Introduction

Correct system timing is a fundamental requirement for the proper operation of a real-time application. The timing definition can dictate how the data information processed during the execution of the application program. The clock implementations vary between devices in the MSP430 family. Each device provides different clock sources, controls and uses. This chapter discusses the clock controls included in the platforms used. The MSP430 4xx family has two general-purpose 16-bit or 8-bit counters and event timers, named Timer_A, Timer_B, and a Basic Timer. The Basic Timer module is only implemented in ‘4xx devices. The 2xx device family also has Timer_A and Timer_B, but the clock signals are provided by the basic clock module+. The timers may receive an internal or external clock. Timer_A and Timer_B also include multiple independent capture and compare blocks, with interrupt capabilities.

Overview

The Real Time Clock (RTC) has a 32-bit counter, to automatically control the clock calendar. This peripheral is present on the MSP430FG461x devices. The application developed in the laboratory *Timers: Lab1 - Memory clock with Basic Timer1* will now be modified to incorporate this module.

Resources

This application (Lab2_Timers.c) is based on the same resources used in the laboratory *Timers: Lab1 - Memory clock with Basic Timer1*. In addition, there is an additional RTC peripheral and two push buttons, SW1 and SW2. The first module works in automatic mode to manage the clock calendar, while the push buttons switch the information displayed on the LCD between the clock and calendar.

Software application organization

The organization of the software is identical to that of laboratory *Timers: Lab1 - Memory clock with Basic Timer1*. The Basic Timer1, LCD and LEDs continue to perform the same functions. They are configured similarly, but with the changes described below.

In routine `main()`, the configurations for RTC and SW1/SW2 are added.

The memory addresses corresponding to the clock calendar values are initialized with the default values, that is zero hours, zero minutes and zero seconds, on August 9, 2008. The RTC is then activated in calendar mode, with the interrupt disabled. This mode affects the Basic Timer1 operation.

The switches SW1 and SW2 are connected to the microcontroller ports P1.0 and P1.1 respectively. Hence, these ports are configured as inputs and their interrupts activated by a high-to-low transition at the input.

System configuration

Real Time Clock configuration

The RTC is configured in calendar mode and enabled. The counting registers provide the values of seconds, minutes, hours, days, day of the week, day of the month, month and year. The registers are stored in BCD format to speed up the data writing process to the LCD. The interrupt for

this peripheral should be disabled (disabling the Basic Timer1 interrupt). Given these objectives:

```
RTCCTL = RTCBCD | RTCHOLD | RTCMODE_3; // BCD mode, RTC and BT disable
```

The RTC operation in calendar mode automatically configures some of the Basic Timer1 features. The content of the bits BTSSEL, BTHOLD and BTDIV of BTCNT register are ignored. Thus, the BTCNT1 and BTCNT2 counters work in cascade. The clock source of the BTCNT1 counter is the ACLK clock signal. The output of the BTCNT1.Q7 counter is selected as the input of the BTCNT2 counter (frequency: ACLK/256). The RTC uses the BTCNT2.Q6 output as clock source (frequency: ACLK/32768).

Basic Timer1 configuration

This peripheral is automatically configured with the RTC in calendar mode. To enable the interrupt once every 0.5 seconds:

```
BTCTL = BT_fCLK2_DIV64; // (ACLK/256)/64 IE2 |= BTIE; // Enable BT interrupt with 0.5 period
```

Ports P1.0 and P1.1 configuration

The switches SW1 and SW2 are connected to ports P1.0 and P1.1 respectively. How should the following registers be configured in order to set just the bits that affect the digital inputs, with high-to-low transition interrupts?

```
P1SEL &= ~0x03; // P1.0 and P1.1 I/O ports P1DIR &= ~0x03; // P1.0 and P1.1 digital inputs P1IFG = 0x00; // Clear P1 flags P1IES &= ~0x03; // high-to-low transition interrupts P1IE |= 0x03; // enable port interrupts
```

Analysis of operation

ISR execution time

Performing similar procedures to those described in laboratory *Timers: Lab1 - Memory clock with Basic Timer1* measure the ISR execution time. What is the value measured?

LCD refresh: _____

The LCD write routines were changed. Taking advantage of storing the data in the BCD format, the division operation can be ignored, resulting in the reduction of execution time of the Basic Timer1 ISR. Is the processing time required to refresh the LCD constant? _____

Measurement of electrical current drawn

The power consumption was discussed in the previous point. The electrical power required by the system during operation is measured by replacing the jumper on the Header PWR1 by an ammeter, which indicates the electric current taken by device during operation.

What is the value read? _____

This example and many others are available on the MSP430 Teaching ROM.

Request this ROM, and our other Teaching Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Laboratory Timers: Lab3 - Memory Clock with Timer_A
Using the MSP-EXP430FG4618 Development Tool and the MSP430FG4618 device implement a Memory Clock with Timer_A.

Laboratory Timers: Lab3 - Memory Clock with Timer_A

Introduction

Correct system timing is a fundamental requirement for the proper operation of a real-time application. The timing definition can dictate how the data information processed during the execution of the application program. The clock implementations vary between devices in the MSP430 family. Each device provides different clock sources, controls and uses. This chapter discusses the clock controls included in the platforms used. The MSP430 4xx family has two general-purpose 16-bit or 8-bit counters and event timers, named Timer_A, Timer_B, and a Basic Timer. The Basic Timer module is only implemented in ‘4xx devices. The 2xx device family also has Timer_A and Timer_B, but the clock signals are provided by the basic clock module+. The timers may receive an internal or external clock. Timer_A and Timer_B also include multiple independent capture and compare blocks, with interrupt capabilities.

Overview

The objective of this laboratory is to build a memory clock similar to the one that was developed using the Basic Timer1, in laboratory *Timers: Lab1 - Memory clock with Basic Timer1*. Timer_A is configured to generate an interrupt once every 100 msec. The ISR manages the memory clock. LED1 and LED2 are used to monitor the operation of the system state.

Resources

This application (Lab3_Timers.c) makes use of Timer_A to generate an interrupt when the value in the TACCR0 unit is reached. The ISR updates the contents of the memory clock variables.

LED1 monitors the system operation, switching state whenever the Timer_A ISR runs. LED2 can be used to monitor the ISR execution time. The contents of the LCD is updated every interrupt. When the ISR finishes, the device returns to low power mode.

Hence, the system resources used by this application are:

- Timer_A;

- I/O ports;

- LCD;

- Interrupts;

- Low power modes.

The default configuration of the FLL+ is used, so all the clock signals required for the operation of the device assume their default values.

Software application organization

The first task is to disable the Watchdog Timer. All the resources needed for the LCD are then configured. Once configured, the LCD is cleared by the execution of the routine `LCD_all_off()`.

The memory clock consists of three global variables: `min`, `sec`, `msec`, of the type `unsigned char`, to store the minutes, seconds and milliseconds respectively of the values elapsed since the beginning of the execution of the application. These variables are initialized with zeros.

The LCD is refreshed at startup to display the initial clock value.

LED2 is used as an indicator of Timer_A ISR execution. The execution time can be monitored using it. In addition, LED1 switches state whenever Timer_A ISR is executed.

Timer_A is configured to generate an interrupt once every 100 milliseconds.

The routine `main()` ends with a global interrupt enable and puts the device into a low power mode, where it waits for the next interrupt.

Timer_A ISR begins by activating LED2, indicating the beginning of execution of the routine and then switches LED1 state. The counters are updated in cascade and their contents are used to update the LCD, through the routines `LCD_msec()`, `LCD_sec()` and `LCD_min()`. The routine ends by switching the state of the clock separation points. Finally, LED2 is turned off.

System configuration

Watchdog Timer

The Watchdog Timer is disabled with the objective of reducing energy consumption, but giving up the protection afforded by it. This peripheral is configured by the WDTCTL register. Its access is protected by a password. The value to disable it:

```
WDTCTL = WDTPW | WDTHOLD; // Stop WDT
```

FLL+ configuration

A 32.768 kHz crystal is applied to the oscillator LFXT1. Since it is possible to select the internal capacitors using software, what is the value to write to the FLL_CTL0 configuration register to select the 8 pF capacitors?

```
FLL_CTL0 |= XCAP18PF; // Set load cap for 32k xtal
```

LED ports configuration

LED1 and LED2 are connected to ports P2.2 and P2.1 respectively. How should they be configured so that just the bits related to these ports have digital output functions?

```
P2DIR |= 0x06; // P2.2 and P2.1 as output
```

How should the P2OUT register be configured so that the application starts with LED1 on and LED2 off?

```
P2OUT |= 0x04; // LED1 on and LED2 off
```

Timer_A configuration

The Timer_A is configured to count until it reaches the value written in the TACCR0 unit. An interrupt is generated when it reaches that value. Which is the interrupt vector to use? _____

Timer_A clock signal is the ACLK without division. What is the value to write in the configuration register?

```
TACTL = TASSEL_1 | MC_1 | ID_0; // ACLK, up mode
```

The TACCR0 capture/compare unit determines the Timer_A counting range. For a 100 msec response, what is the value to write in the register?

```
TACCR0 = 3268; // this count corresponds to 100 msec
```

The interrupt is configured in TACCR0 capture/compare unit. What is the value to write to the following register?

```
TACCTL0 = CCIE; // TACCR0 interrupt enabled
//*****************************************************
********* // Timer A ISR
//*****************************************************
********* #pragma vector=TIMERA0_VECTOR
__interrupt void TimerA0_ISR (void) { P2OUT
|=0x02; // LED1 turn on P2OUT ^=0x04; // LED2
toogle msec++; LCD_msec(); if (msec == 10) { msec
= 0; sec++; LCD_sec(); if (sec == 60) { sec = 0;
min++; LCD_min(); if (min == 60) { min = 0; } } }
if (sec & 0x01) // toogle clock dots { P3_DOT_ON;
P5_DOT_ON; } else { P3_DOT_OFF; P5_DOT_OFF; }
P2OUT & =~ 02; // LED1 turn off }
```

Low power mode

```
BIS_SR(LPM3_bits + GIE); // LPM3 with interrupts
enable
```

Analysis of operation

ISR execution time

Performing similar procedures to those described in laboratory *Timers: Lab1 - Memory clock with Basic Timer1* measure the ISR execution time. What is the value measured?

LCD refresh: _____

The LCD write routines were changed. Taking advantage of storing the data in the BCD format, the division operation can be ignored, resulting in the reduction of execution time of the Basic Timer1 ISR. Is the processing time required to refresh the LCD constant? _____

Measurement of electrical current drawn

The power consumption was discussed in the previous point. The electrical power required by the system during operation is measured by replacing the jumper on the Header PWR1 by an ammeter, which indicates the electric current taken by device during operation.

What is the value read? _____

This example and many others are available on the MSP430 Teaching ROM.

Request this ROM, and our other Teaching Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Laboratory Timers: Lab4 - Buzzer tone generator
Using the MSP-EXP430FG4618 Development Tool and the MSP430FG4618 device implement a Buzzer tone generator.

Laboratory Timers: Lab4 - Buzzer tone generator

Introduction

Correct system timing is a fundamental requirement for the proper operation of a real-time application. The timing definition can dictate how the data information processed during the execution of the application program. The clock implementations vary between devices in the MSP430 family. Each device provides different clock sources, controls and uses. This chapter discusses the clock controls included in the platforms used.

The MSP430 4xx family has two general-purpose 16-bit or 8-bit counters and event timers, named Timer_A, Timer_B, and a Basic Timer. The Basic Timer module is only implemented in ‘4xx devices. The 2xx device family also has Timer_A and Timer_B, but the clock signals are provided by the basic clock module+.

The timers may receive an internal or external clock. Timer_A and Timer_B also include multiple independent capture and compare blocks, with interrupt capabilities.

Overview

The purpose of this laboratory is to build a sound generator using Timer_B. The PWM signal produced by this peripheral drives the buzzer, producing a sequence of musical notes at regular time intervals. At the same time, LED1 and LED2 switch state alternately. The volume of sound produced by the buzzer can be controlled by push buttons SW1 and SW2.

Resources

The implementation of this application (Lab4_Timers.c) requires the production of specific frequency signals corresponding to musical notes. For each frequency, the duty-cycle can be modified in order to control the volume of sound produced. This task is carried out using Timer_B and one of its compare units. The buzzer is operated by Port P3.5, configured to work in its special function as TB4 compare unit output. This output corresponds to the TBCCR4 output compare unit.

The push buttons SW1 and SW2 are connected to ports P1.0 and P1.1 respectively. An interrupt is generated when either of these buttons are pressed. The duty-cycle of the generated note is modified in response.

Basic Timer1 is configured to generate an interrupt once every second. The interrupt service routine updates the musical notes produced by the buzzer, which are stored in an array.

LED1 and LED2 are driven from P2.2 and P2.1 respectively, and their state is switched alternately once every second.

The module FLL+ is configured to a 7.995392 MHz frequency, for the MCLK and SMCLK clock signals.

The resources used by the application are:

- Timer_B;

- Basic Timer1;

- I/O ports;

- FLL+;

- Interrupts.

Software application organization

The application consists of the routine `main()`, which is used to configure all system resources, before entering into a standby mode, waiting for one of two interrupts.

This routine starts by disabling the watchdog timer and starting the module FLL+ to produce the desired clock signals of the correct frequency for the SMCLK and MCLK. Then, the Basic Timer1 and Timer_B are configured in order to perform the desired functions.

The ports connected to the LEDs, buttons and buzzer are then initialized.

Finally, the interrupts are activated, and the application waits for the execution of one of two interrupts.

The Basic Timer1 interrupt executes at a frequency of once every second. When this interrupt is occurs, it begins by switching the state of LED1 and LED2. Afterwards, it accesses the memory to fetch the next musical note to be performed. The routine ends with memory pointer management.

The Port 1 ISR begins by evaluating the source of the interrupt. The sound volume is reduced if the button SW1 is pressed. The sound volume is increased if button SW2 is pressed.

System configuration

Timer_B

It is the responsibility of Timer_B to produce the PWM signal that activates the Buzzer. Timer_B counts until the value contained in the TBCCR0 register is reached. It does not generate an interrupt, and must be sourced by SMCLK clock signal:

`TBCTL = TBSSEL_2 | CNTL_0 | TBCLGRP_0 |MC_1 | ID_0;`

Each PWM signal produced by Timer_B corresponds to a musical note. The relationship between the frequency and the musical note is given in Table 1.

Note	SI0	DO	RE	MI	FA	SOL	LA	SI	DO2
Freq [Hz]	503	524	587	662	701	787	878	1004	1048

Timer_B has a frequency clock input equal to 7.995392 MHz.

The value to write in the TBCCR0 register in order to generate the desired frequency is:

```
// TBCCR0 value of the musical notes
#define SI0 15895
#define DO 15258
#define RE 13620
#define MI 12077
#define FA 11405
#define SOL 10159
#define LA 9106
#define SI 7963
#define DO2 7629

TBCCTL4 = OUTMOD_3; // CCR4 interrupt enabled
TBCCR4 = space[0]/2;
```

Timer_A configuration

```
TACTL = TASSEL_2 |MC_2 | ID_0 | TAIE; // SMCLK, continuous mode
up to 0xffff

TACCTL1 = CM1 | CCIS_0 | CAP | CCIE;// Capture on rising edge,
```

```
Cap mode,
                                        // Cap/Com int. enable,
TACCR1 input signal selected

//*************************************************************
// Timer A ISR
//*************************************************************
#pragma vector=TIMERA1_VECTOR
__interrupt void TimerA1_ISR (void)
{
 switch (TAIV)
 {
  case TAIV_TACCR1:
  if (capture == 0){
  T1 = TACCR1;
  flag = 1;
  capture = 1;
 }
 else {
  if (flag == 1) {
   T2 = TACCR1;
   if (T2 > T1)
    T = T2-T1;
   }
   else{
    TAR = 0;
   }
   capture = 0;
   flag = 0;
  }
 break;


case TAIV_TACCR2:
    break;

    case TAIV_TAIFG:
    tick++;
    if (tick == 60){
    LCD_freq();
    tick = 0;
    }
    if (flag == 1)flag = 0;


break;
```

```
default:
    break;
    }
}
```

Basic Timer1

The Basic Timer1 generates an interrupt once every second. It uses two counters in series, where the BTCNT2 counter input uses the BTCNT1 counter output divided by 256. The BTCNT1 counter input is the ACLK clock signal with a frequency of 32.768 kHz.

If BTCNT2 counter selected output is divided by 128, what is the time period associated with the Basic Timer1 interrupt? _____

What are the values to write to the configuration registers?

```
BTCTL = BTDIV | BT_fCLK2_DIV128; // (ACLK/256)/128
IE2 |= BTIE; // enable BT interrupt

//*********************************************************
// Basic Timer ISR. Run with 1 sec period
//*********************************************************
#pragma vector=BASICTIMER_VECTOR
__interrupt void basic_timer_ISR(void)
{
unsigned int read_data; // read data from file , frequency in
kHz

P2OUT^=0x06; // toogle LED1 and LED2
counter++;

if (counter == 5){
    counter = 0;
     read_data = 200;
     TBCCR0 = 7995392/read_data;
     TBCCR4 = TBCCR0/2;
    }
}
```

I/O Ports configuration

```
// SW1 and SW2 configuration (Port1)
P1SEL &= 0x00; // P1.0 and P1.2 I/O
P1DIR &= 0x00; // P1.0 and P1.2 as inputs
P1IFG = 0x00;
P1IES &= 0xFF // high-to-low transition interrupt
P1IE |= 0xFF; // enable port interrupts

// LED1 and LED2 configuration (Port2):
P2DIR |= 0x06; // P2.2 and P2.1 as outputs
P2OUT = 0x04; // LED1 on and LED2 off

// Buzzer port configuration (Port3)
P3SEL |= 0x20; // P3.5 as special function
P3DIR |= 0x20; // P3.5 as digital output
```

FLL+ configuration

```
FLL_CTL0 |= DCOPLUS + XCAP18PF; //DCO+ set,freq=xtal*D*N+1
SCFI0 |= FN_4; // x2 DCO freq, 8MHz nominal DCO
SCFQCTL = 121; // (121+1) x 32768 x 2 = 7.99 MHz
```

Analysis of operation

System clocks inspection

The MCLK, SMCLK and ACLK system clocks are available at ports P1.1, P1.4 and P1.5 respectively. These ports are located on the SW2, RESET_CC and VREG_EN lines, which are available on the H2 Header pins 2, 5 and 6. All these resources are available because the Chipcon RF module is not installed and SW2 is not used.

Using the Registers view, set bits 1, 4 and 5 of P1SEL and P1DIR registers to choose the secondary function of their ports, that is, configured as outputs. Connect an oscilloscope probe at these positions to monitor the clock signals.

What are the values measured for each of the system clocks?

ACLK: _____

SMCLK: _____

MCLK: _____

TBCCR4 unit output frequency

With the help of an oscilloscope, it is possible to evaluate the operation of the application. Alternatively, it is possible to listen to the sound produced. By removing jumper JP1 and connecting the oscilloscope to this pin, it is possible to view the PWM signal produced by the microcontroller. The duty-cycle can be reduced or increased by pressing the push buttons SW1 and SW2.

Port P1 interrupt source decoding

All Port P1 interrupt lines share the same interrupt vector. The decoding is done through the P1IFG register.

This process can be observed by entering a breakpoint at the first line of the ISR code.

Execute the application.

The application's execution is suspended at the breakpoint by pressing either button SW1 or SW2. From this point onwards, run the lines of code step-by-step and observe changes in the register values.

Measurement of electrical current drawn

The power consumption was discussed in the previous point. The electrical power required by the system during operation is measured by replacing the jumper on the Header PWR1 by an ammeter, which indicates the electric current taken by device during operation.

What is the value read? _____

This example and many others are available on the MSP430 Teaching ROM.

Request this ROM, and our other Teaching Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Laboratory Timers: Lab5 - Frequency measurement
Using the MSP-EXP430FG4618 Development Tool and the MSP430FG4618 device perform a frequency measurement.

Laboratory Timers: Lab5 - Frequency measurement

Introduction

Correct system timing is a fundamental requirement for the proper operation of a real-time application. The timing definition can dictate how the data information processed during the execution of the application program. The clock implementations vary between devices in the MSP430 family. Each device provides different clock sources, controls and uses. This chapter discusses the clock controls included in the platforms used.

The MSP430 4xx family has two general-purpose 16-bit or 8-bit counters and event timers, named Timer_A, Timer_B, and a Basic Timer. The Basic Timer module is only implemented in ‘4xx devices. The 2xx device family also has Timer_A and Timer_B, but the clock signals are provided by the basic clock module+.

The timers may receive an internal or external clock. Timer_A and Timer_B also include multiple independent capture and compare blocks, with interrupt capabilities.

Overview

This laboratory implements an application (Lab5_Timers.c) designed to measure a PWM signal frequency. If a signal generator is not available, the microcontroller generates a PWM signal based on the frequencies stored in a file. The frequencies generated are read and updated with a fixed time period using the features of **CCE**. The measured value is shown on the LCD in Hz.

Resources

The module FLL+ is configured to a frequency of 7.995392 MHz for the MCLK and SMCLK clock signals. This application performs the two tasks simultaneously.

On the one hand, it generates a PWM signal with a frequency of 200 Hz and a duty cycle of 50%. Alternatively, the PWM signal frequency can be read from a file using a breakpoint. This function is performed by Timer_B, using the compare unit to generate the PWM signal.

The time period between two consecutive PWM signals low-to-high transitions is measured by Timer_A. The capture unit of this timer is configured to collect the Timer_A counter register's contents when a PWM signal low-to-high transition is detected at its input.

The Basic Timer1 generates an interrupt once every second. The ISR updates the PWM signal frequency generated by the Timer_B. If you choose to use this feature, a breakpoint associated with this ISR execution allows reading a file with the value of the frequency that will be generated.

The microcontroller's ports are configured in order that the PWM signal generated by Timer_B through the TBCCR4 compare unit available at Port P3.5/TB4 can be connected to the Port P1.2/TA1 of the Timer_A TACCR1 capture unit. If you plan to use this feature, these pins must be connected together. Port P3.5 pin is available on Header 7 pin 6, while the Port P1.2 pin is available on Header H2 pin 3.

Ports P2.1 and P2.2 are used to monitor the state of the LED2 and LED1, respectively.

The resources used by the application are:

- Timer_A;

- Timer_B;

- Basic Timer1;

- I/O ports;

- FLL+;

- Interrupts.

Software application organization

The software structure allows various tasks to be performed simultaneously. The routine `main()` is responsible for configuring all the resources used by the application. Once started, the application enables all the interrupts and waits for an interrupt request.

There are two routines that separately service the two possible interrupts. The routine `TimerA1_ISR()` services interrupts required by the Timer_A overflow and by the **TACCR1** capture unit. For every interrupt caused by a **TACCR1** capture, the value collected in the **TACCR1** register is stored in **T1**, if it is the first low-to-high transition, or stored in **T2** if it is the second low-to-high transition. This sequence is controlled by the variable capture. The variable flag is used to flag the measurement process. This process occurs between the capture of the first low-to-high transition and the second transition. The counting of clock pulses is done by Timer_A, in the time interval between the **T1** and **T2** acquisition, assigned to the variable **T**. The process is synchronized when Timer_A overflows, restarting the measurement process. The LCD is refreshed once every 0.5 seconds with the latest measured frequency value, using the control variable control tick that corresponds to 0.5 seconds.

The routine `basic_timer_ISR()` services the interrupt produced by Basic Timer1 once every second. This routine begins by switching the state of LED1 and LED2. In addition, it updates the Timer_B counting period. The variable `read_data` allows the counting period to be changed.

System configuration

Basic Timer1

Basic Timer1 generates an interrupt once every second. Use the two counters in series, where the BTCNT2 counter input is selected as the BTCNT1 counter output divided by 256. The BTCNT1 counter input is the ACLK clock signal with a frequency of 32.768 kHz.

If BTCNT2 counter selected output is divided by 128, what is the time period associated with the Basic Timer1 interrupt? _____

The values written to the configuration registers are:

```
BTCTL = BTDIV | BT_fCLK2_DIV128; // (ACLK/256)/128
IE2 |= BTIE; // Enable BT interrupt with 1 sec
period
```

Timer_B

The TBCCR4 compare unit is used to generate the PWM signal. The set/reset compare mode is used.

The values written to the configuration registers are:

```
TBCTL = TBSSEL_2 | CNTL_0 | TBCLGRP_0 |MC_1 |
ID_0;
// SMCLK, continuous mode
TBCCTL4 = OUTMOD_3; // CCR4 output mode 3
(set/reset)
```

The TB4 PWM output signal has a frequency X, with a 50% duty-cycle. The SMCLK clock signal is used as input of Timer_B.

The values written to the configuration registers are:

```
TBCCR0 = 39977;// Output 200 Hz signal with 50%
duty cycle
TBCCR4 = TBCCR0/2;
```

What the largest and lowest generated frequency?

Maximum frequency value: _____

Minimum frequency value: _____

Timer_A

Timer_A is sourced by the SMCLK clock signal. It counts to the value 0xFFFF, in continuous mode. An interrupt is generated when the TAR counter overflows. What is the value to write to its configuration register?

```
TACTL = TASSEL_2 |MC_2 | ID_0 | TAIE; // SMCLK
// up mode to 0xFFFF
```

The capture unit captures the TAR register value to the TACCR1 register when it detects a low-to-high transition at the TA1 input. What is the value to write to the configuration register?

```
TACCTL1 = CCIS_0 | CAP | CCIE;
// Capture on rising edge,
// TACCR1 input signal selected,
// Capture mode,
// Capture/compare interrupt enable.
```

Determine the maximum and minimum frequency values detected. Note that these values do not take into account the execution time of the application. The PWM signals should be applied at frequencies well below the maximum value determined.

Maximum frequency value: _____

Minimum frequency value: _____

The TACCR1 capture unit is configured to generate an interrupt when it detects a low-to-high transition. What is the value to write to the configuration register?

`TACCTL1 |= CM1`

Ports P3.5/TB4 and P1.2/TA1 configuration

These ports perform special functions. Thus, the Port P3.5 is configured as an output, selected by the special function TB4, with the values:

```
// TB4 configuration (Port3)
P3SEL = 0x20; // P3.5 as special function (TB4)
P3DIR = 0x20; // P3.5 as output
```

The Port P1.2 is configured as input, with the special function TA1, using the values:

```
// TA1 (TACCR1) configuration (Port1)
P1SEL = 0x04; // P1.2 as special function (TA1)
P1DIR = 0x00; // P1.2 as input
```

I/O Ports configuration:

```
// SW1 and SW2 configuration (Port1)
P1SEL &= 0x00; // P1.0 and P1.2 I/O
P1DIR &= 0x00; // P1.0 and P1.2 as inputs
P1IFG = 0x00;
P1IES &= 0xFF // high-to-low transition interrupt
```

```
P1IE |= 0xFF; // enable port interrupts

// LED1 and LED2 configuration (Port2):
P2DIR |= 0x06; // P2.2 and P2.1 as outputs
P2OUT = 0x04; // LED1 on and LED2 off

// Buzzer port configuration (Port3)
P3SEL |= 0x20; // P3.5 as special function
P3DIR |= 0x20; // P3.5 as digital output
```

Analysis of operation

Run the application using the frequency generator based on Timer_B

Without a frequency generator, the Timer_B generates a PWM signal at the TBCCR4 unit output that can be fed back to Timer_A TACCR1 capture unit input. These two pins must therefore be connected together. By default, the PWM signal frequency is 200 Hz. Add a breakpoint at the line of code belonging to the Basic Timer1 ISR to modify this value.

`TBCCR0 = 7995392/read_data;`

If the variable `read_data` has the value 200, it will generate a 200 Hz frequency. The value of this variable can be changed by associating a breakpoint to that line of code. Before the line of code is executed, the value of the data file is read and assigned to the variable `read_data`. The signal will oscillate at the desired frequency, loading the value in TBCCR0. The breakpoint configuration is as follows:

- Action: read data from file

- File: address of the data file (example in freq.txt)

- Wrap Around: activate this option to restart reading at the beginning

- Start address: &read_data

- Length: 1 in order to read a value from the file each time

Run the application using a frequency generator

The operation of the application can be verified using a frequency generator. The generator should generate a PWM signal with voltage and frequency values compatible with the device's input (2.5 to 3.3 volts).

Observe the measured frequency

The PWM signal applied to the TA1 input can be viewed using an oscilloscope, connected to pin 3 of Header 2. Perform this task and confirm the values present at the LCD.

Measurement of electrical current drawn

The power consumption was discussed in the previous point. The electrical power required by the system during operation is measured by replacing the jumper on the Header PWR1 by an ammeter, which indicates the electric current taken by device during operation.

What is the value read? _____

This example and many others are available on the MSP430 Teaching ROM.

Request this ROM, and our other Teaching Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Laboratory LCD controller: Lab1 - LCD message display
Using the MSP-EXP430FG4618 Development Tool and the MSP430FG4618 device present a message on the LCD Basic Timer1.

Laboratory LCD controller: Lab1 - LCD message display

Introduction

This hands-on laboratory consists of configuring the LCD_A controller of the MSP430FG4618 device of the Experimenter's board to display a message on the LCD display. This laboratory has been developed for Code Composer Essentials version 3 software development tool only.

Overview

This laboratory will explore the LCD_A controller of the MSP430FG4618 device included on the Experimenter's board. This application (Lab1_LCD.c) demonstrates the activation of various LCD segments.

Resources

The Experimenter's board uses a LCD, which does not have its own controller. The operation is controlled by MSP430FG4618.

The interface between these two components is described in the Experimenter's Board datasheet slau213a.pdf

It is also recommended that the LCD datasheet be read.

Based on this information, it is possible to define the values to write to each of the memory registers to turn on the desired segments, or to set several of them, as is the case with numbers. The definitions are listed in LCD_defs.h.

From analysis of the Experimenter's Board schematics, it can be seen that there is a 10 µF between the LCDCAP pin and ground, which means it is possible to use the charge pump.

The segments shared by the I/O function are not used by the LCD, being connected to the segments S4 to S25. The four lines COM0, COM1, COM2, and COM3 are used. The last three lines are shared by ports P5.2, P5.3 and P5.4, respectively. The LCD is operated in 4-mux mode.

The pins R03, R13, R23 and LCDCAP\R33 are used to provide the V5, V4, V3, V2 and V1 (V_{LCD}) voltages, using an external resistor network. They are available at Header H5.

In the current Experimenter's Board configuration, it is possible to select the AV_{ss} or charge pump to provide the V1 (V_{LCD}), V2, V3, V4 and V5 voltages. These voltages are only generated when LCD_A module and the ACLK clock are active. This allows the use of low power mode 3 (LPM3).

Timer_A, together with the TACCR0 unit are used to generate an interrupt once every second. LED1 and LED2 are switched at each Timer_A interrupt.

The push button SW1 is used to change the value of voltage generated by the charge pump. The push button SW2 is used to change the LCD frequency.

Software application organization

The application starts by configuring the Ports P5.2, P5.3, P5.4 to special function COM1, COM2 and COM3, respectively. The function of COM0 is not shared with the digital I/O functions.

Then, the pins with multiplexed functions are selected to perform the functions necessary to control the LCD segments.

The LCD_A control register and the voltage configuration register are also configured.

There then follows the execution of the LCD clear routine `LCD_all_off()`, which ensures that all segments of the LCD are off.

Timer_A is configured with its TACCRO unit to generate an interrupt once every second. The ISR generates the memory clock with **msec**, **sec** and **min**, and also connects/disconnects the remaining LCD symbols.

The port pins P2.1 and P2.2 drive LED2 and LED1, respectively. Hence, these ports are configured as digital outputs.

Push buttons SW1 and SW2 have the capacity to generate an interrupt through a change at ports P1.0 and P1.2 respectively. The interrupt ISR, after decoding its source, modifies the LCD operation frequency or modifies the VLCD voltage.

Finally, all the interrupts are activated and the system enters low power mode LPM3.

System configuration

LCD_A interface with the LCD configuration

Select the function COM1, COM2 and COM3. What is the value to write to these registers?

```
P5DIR |= 0x1E; // Ports P5.2, P5.3 and P5.4 as
outputs
P5SEL |= 0x1E; // Ports P5.2, P5.3 and P5.4 as
special function (COM1, COM2 and COM3)
```

The LCD segments are controlled by the S4 to S25 LCD memory segments. Activate these segments by writing to correct value to the following register:

```
LCDAPCTL0 = LCDAPCTL0 = LCDS24 | LCDS20 | LCDS16 |
LCDS12 | LCDS8 | LCDS4; // Enable S4 to S25
```

LCD operation frequency

The LCD is to operate in 4-mux mode, with a 30 Hz to 100 Hz refresh frequency. It uses the following equation to determine the LCD operation frequency, f_{LCD}:

$f_{LCD} = 2 \times mux \times f_{frame}$

Choose the frequency that provides greatest energy savings.

LCD_A configuration

The LCD_A module is to be activated in 4-mux mode from a 32768 Hz clock. What value should be written to the following register?

```
LCDACTL = LCDFREQ_192 | LCD4MUX; // (ACLK =
32768)/192
    // and 4-mux LCD
LCDACTL |= LCDSON | LCDON; // LCD_A and Segments
on
```

Use the charge pump to internally generate all the voltages necessary for the operation of the LCD, using a bias 1/3. What is the value to write to the register?

```
LCDAVCTL0 = LCDCPEN; // Charge pump enable
```

The charge pump generates a LCD voltage of 3.44 volts. Configure the following register:

```
LCDAVCTL1 = VLCD_3_44; // VLCD = 3.44 V
```

Timer_A configuration

The Timer_A generates an interrupt once every second. It uses the TACCR0 unit. Configure the following registers:

```
TACCTL0 = CCIE; // TACCR0 interrupt enabled
TACCR0 = 3268; // this count correspond to 1 msec
TACTL = TASSEL_1 | MC_1 | ID_0; // ACLK, up mode
```

Output ports configuration

Configure the ports connected to LED1 and LED2 in order to make one of them active and the other inactive at system start up:

```
P2DIR |= 0x06; // P2.1 and P2.2 as output
P2OUT |= 0x04; // LED2 off and LED1 on
```

Input ports configuration

The push buttons SW1 and SW2 generate an interrupt on a low-to-high transition. Configure the necessary registers:

```
P1DIR &= ~0x03; // P1.0 and P1.1 digital inputs
P1IES |= 0x03; // low-to-high transition
interrupts
P1IE |= 0x03; // enable port interrupts
```

Analysis of operation

Compile the project, load it into microcontroller's memory and execute the application. For each value of the operating frequency and voltage generated by the charge pump, measure the electrical current consumption. Draw a graph of these results and draw conclusions concerning the energy consumption.

This example and many others are available on the MSP430 Teaching ROM.

Request this ROM, and our other Teaching Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Laboratory Signal Acquisition: Lab1 - SAR ADC10 conversion
Using the eZ430-RF2500 Development Tool use ADC10 to perform a single temperature sample on channel A10 (on-chip temperature sensor) each minute during 1 hour.

Laboratory Signal Acquisition: Lab1 - SAR ADC10 conversion

Introduction

This laboratory gives examples of the uses of the ADC types available in the hardware development kits. A different laboratory is developed for each kit, taking into account that both the ADC10 and the SD16_A laboratories implement a temperature data logger. The ADC12 laboratory also uses operational amplifiers to perform the analogue signal conditioning.

Overview

This laboratory (Lab1_ADC.c) implements a temperature data logger using the hardware kit's integrated temperature sensor. The device is configured to perform an acquisition each minute for one hour. Each temperature's (ºC) value is transferred to flash info memory segment B and C. When the microcontroller is not performing any task, it enters into low power mode.

Resources

The ADC10 module uses V_{REF+} = 1.5 V as the reference voltage.

It is necessary to configure the ADC10 to use the integrated temperature sensor (A10) as an input. Timer_A generates an interrupt once every second that starts conversion in the ADC10. At the end of a conversion, an interrupt is requested by the converter and the temperature value is written to flash memory.

The voltage value is converted into temperature following the equation provided in ADC10 section of the MSP430 User's Guide <slau144e.pdf>. After transferring the value to the flash memory, the system returns to low power mode LPM3.

The resources used by the application are:

- ADC10;

- Timer_A;

- Ports I/O;

- Interrupts;

- Low power mode.

Software application organization

The application starts by stopping the Watchdog Timer.

The system checks for calibration constants on info memory segment A. The CPU execution will be trapped if it does not find this information.

Digitally controller oscillator (DCO) is set to 1 MHz, providing clock source for MCLK and SMCLK, while the Basic Clock System+ is configured to set ACLK to 1.5 kHz.

Controller's flash timing is obtained from MCLK divided by three to comply with the device specifications.

Port P1.0 is configured as an output and will blink the once LED every second.

The ADC10 is configured to use the input channel corresponding to the on-chip temperature sensor (channel A10). The configuration includes the activation of the internal reference voltage V_{REF+} = 1.5 V and the selection

of ADC10OSC as clock signal. The converter is configured to perform a single-channel single-conversion. At the end of conversion, an interrupt is requested.

The Timer_A is configured to generate an interrupt once every second. ACLK/8 is selected as the clock signal using the VLOCLK as clock source and will count until the TACCR0 value is reached (in up mode). The system then enters into low power mode and waits for an interrupt.

Flash memory pointers and interrupt counters are initialized. The Timer_A ISR increments the variable **counter** and when this variable reaches the value 60 (1 minute), the software start of conversion is requested. At the end of this ISR, the system returns to low power mode.

When the ADC10 ends the conversion, an interrupt is requested. While variable **min** is lower than 60, the temperature is written to flash memory. The memory pointer is increased by two (word). When `min = 60`, the system stops operation.

System configuration

DCO configuration

Adjust the DCO frequency to 1 MHz by software using the calibrated DCOCTL and BCSCTL1 register settings stored in information memory segment A.

```
if (CALBC1_1MHZ == 0xFF || CALDCO_1MHZ == 0xFF)
    {
      while(1); // If calibration constants erased
      // do not load, trap CPU!!
    }

DCOCTL = CALDCO_1MHZ; // Set DCO to 1 MHz
```

Basic Clock module+ configuration

Set MCLK and SMCLK to 1 MHz. Use the internal very low power VLOCLK source clock to ACLK/8 clock signal as low frequency oscillator (12 kHz):

```
BCSCTL1 = DIVA_3; // ACLK = 1.5 kHz
BCSCTL3 = LFXT1S_2; // Set VLOCLK (12 kHz)
```

ADC10 configuration

The ADC10's input channel is the integrated temperature sensor (A10) and it uses the signal V_{REF+} (1.5 V) as reference voltage. The ADC10 clock source is ADC10OSC, the clock signal being ADC10CLK/4. Configure the ADC10 sample-and-hold time: 64xADC10CLKs, to perform a single-channel single-conversion and enable its interrupts. What are the values to write to the configuration registers?

```
ADC10CTL1 = INCH_10 + ADC10DIV_3; // Temp Sensor
(A10),
// ADC10CLK/4,
// clock source: ADC10OSC
ADC10CTL0=SREF_1 + ADC10SHT_3 + REFON + ADC10ON
+ADC10IE;
// Internal reference voltage Vref+ = 1.5 V
// ADC10 sample-and-hold time: 64 x ADC10CLKs
// Reference-generator voltage = 1.5 V
// ADC10 on + ADC10 interrupt enable

//**************************************************
*********
// ADC10 Interrupt Service Routine
//**************************************************
*********
#pragma vector=ADC10_VECTOR
__interrupt void ADC10ISR(void)
```

```
{
  unsigned int temperature;


if (min <= 60)
    {
      temperature = ((ADC10MEM - 673) * 423) /
1024;
      write_int_flash(memo_ptr,temperature);
      memo_ptr += 2;
    }
    else
    {
      _NOP();
    }
}
```

Timer_A configuration

Configure Timer_A register to enable an interrupt once every second. Use the ACLK clock signal as the clock source. This timer is configured in up counter mode in order to count until the TAR value reaches the TACCR0 value.

```
TACCTL0 = CCIE; // TACCR0 interrupt enabled
TACCR0 = 1500; // this count corresponds to 1 sec
TACTL = TASSEL_1 | MC_1 | ID_0;// ACLK, up mode to
TACCR0

//**************************************************
**********
// Timer_A Interrupt Service Routine
//**************************************************
**********
#pragma vector=TIMERA0_VECTOR
```

```
__interrupt void TimerA0_ISR (void)
{
    counter++;
    P1OUT ^= 0x01; // LED toogle

    if (counter == 60)
    {
      min++;
      counter = 0;
      ADC10CTL0 |= ENC + ADC10SC; //
Sampling/Conversion start
    }
}
```

Analysis of operation

After compiling the project, start the debug session and before running the application, put a breakpoint at the line of code with the `_NOP()` instruction. Go to breakpoint properties and set action to **Write data to file**. Name the file as **Temp.dat** and define the **data format** as **integer**. The data starts at address `0x01040`, with a length of `3C`. Run the application and let the temperature data logger acquire the values for 1 hour. Use a heater or a fan to force temperature variations during the measurement period. When execution reaches the breakpoint, the file will be available in your file system. Construct a graph in Excel or a similar tool, in order to plot the temperature variation obtained by the data logger.

This example and many others are available on the MSP430 Teaching ROM.

Request this ROM, and our other Teaching Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Laboratory Signal Acquisition: Lab2 - SAR ADC12 conversion
Using the MSP-EXP430FG4618 Development Tool and the MSP430FG4618 device explore the ADC12 and OA modules. The test voltage is generated in the DAC12 module (channel 0) modifying the DAC12_ODAT register's value. The analogue voltage is amplified by the OA module. Afterwards this signal is applied to ADC12 input to be converted. Compare the DAC12_ODAT and the ADC12MEM0 values.

Laboratory Signal Acquisition: Lab2 - SAR ADC12 conversion

Introduction

This laboratory gives examples of the uses of the ADC types available in the hardware development kits. A different laboratory is developed for each kit, taking into account that both the ADC10 and the SD16_A laboratories implement a temperature data logger. The ADC12 laboratory also uses operational amplifiers to perform the analogue signal conditioning.

Overview

This laboratory (Lab2_ADC.c) examines the ADC12 and OA modules using the MSP-EXP430FG4618 Development Tool (MSP430FG4618 device). The test voltage is generated by the DAC12 channel 0, available in DAC12_ODAT register. The analogue signal is conditioned by the OA module (amplitude change), configured as non-inverting operational amplifier. Afterwards, this signal is applied to the ADC12 input to be converted. Compare the DAC12_ODAT and the ADC12MEM0 values.

Resources

The DAC12 module uses the same internal reference voltage as the ADC12 module (V_{REF+} = 2.5 V).

The OA module is configured as Non-inverting PGA with unity gain. The Non-inverting input is the DAC0 internal while the output is connected to internal/external A1 of the ADC12. The ADC12 sample-and-hold time is configured to be 64 ADC12CLK cycles. It performs a single-channel, single-conversion using ADC12OSC/1 as the clock source.

The resources used by the application (following the signal modification steps) are:

- DAC12;

- OA;

- ADC12;

- Timer_A;

- Interrupts.

Software application organization

The laboratory is organized following its working flow chart:

- Peripheral initialization phase, finishing with the MSP430 in LPM3;

- ISR phase, consisting of a Timer_A overflow service routine that triggers a new ADC12 conversion and it is responsible by the end of conversion.

The application starts by stopping the Watchdog Timer.

The system clock is configured by the FLL+ at 4.199304 MHz (128 x 32768Hz).

The DAC12 module is configured to present a null voltage (0 V) at the output. It uses the ADC12 internal 2.5 V reference voltage. The DAC12’s output is configured with 12-bit resolution, in straight binary. DAC12 uses the full-scale output with a Medium speed/current.

The OA module is configured as non-inverting PGA, the input signal (DAC0 internal) being in the rail-to-rail range. The output of the OA is connected to internal/external A1.

The ADC12 is configured to perform a single-channel (channel A1), single-conversion. The configuration includes the activation of the same internal reference voltage as the DAC12. The ADC12 clock source is ADC12OSC, with the sample-and-hold time selected as 64 ADC12CLK cycles.

The Timer_A is configured to use the ACLK as the clock source. It will count in continuous mode (TACCR0 counts up to 0FFFFh) and generate an interrupt to update the ADC12MEM. When the interrupt is serviced, the MSP430 enters into LPM3.

System configuration

ADC12 configuration:

The ADC12 module is configured in order to have the following characteristics:

- Single-channel, single-conversion operation;

- Uses the internal signal V_{REF+} (2.5 V) as reference voltage;

- The sample-and-hold time must be 64 ADC12CLK cycles;

- The conversion result must be available on ADC12MEM0;

- The sample-and-hold clock source is defined by software.

```
ADC12CTL0 |=
SHT02|REF2_5V|REFON|ADC12ON|ENC|ADC12SC;
    //SHT1x (Sample-and-hold time) = 0000b -> N/A
    //SHT0x (Sample-and-hold time) = 0010b -> 64
```

```
ADC12CLK
    //MSC (Multiple sample and conversion) = 0b ->
N/A
    //REF2_5V (Reference generator voltage) = 1b -
> 2.5 V
    //REFON (Reference generator on) = 1b ->
Reference on
    //ADC12ON (ADC12 on) = 1b -> ADC12 on
    //ADC12OVIE (overflow-int. enable) = 0b ->
disabled
    //ADC12TOVIE (conversion-time-overflow int
enable) = 0b
    // -> disabled
    //ENC (Enable conversion) = 0b -> enable
configuration
    //ADC12SC (Start conversion) = 1b -> Start
conversion

ADC12CTL1 = CSTARTADD_0; // Start MEM0, TB1, Rpt
Sing.
    //CSTARTADDx (Conv. start address.) = 0000b ->
ADC12MEM0
    //SHSx (Sample-and-hold source) = 00b ->
ADC12SC bit
    //SHP (Sample-and-hold pulse-mode select) = 0b
    // -> SAMPCON is sourced from the sample-input
signal
    //ISSH (Invert signal S-H) = 0b -> not
inverted
    //ADC12DIVx (ADC12 clock divider) = 000b -> /1
    //ADC12SSELx (ADC12 clock source) = 00b ->
ADC12OSC
    //CONSEQx (Conversion sequence mode) = 00b ->
Single-
    // channel, single-conversion
    //ADC12BUSY (ADC12 busy) = xb -> read only
```

The ADC12 module operates with reference voltages: $V_{R+} = V_{REF+}$ and $V_{R-} = AV_{SS}$. The channel selected to perform the analogue-to-digital conversion is channel A1. This channel is internally connected the OA0's output.

```
ADC12MCTL0 = INCH_1 | SREF_1;
    //EOS (End of sequence) = 0b -> Not Used
    //SREFx (Select ref.) = 001b -> VR+=VREF+/VR-
=AVSS
    //INCHx (Input channel select) = 0001b -> A1
```

DAC12 configuration:

```
DAC12_0DAT = 0x00; // DAC_0 output 0V

DAC12_0CTL = DAC12IR | DAC12AMP_5 | DAC12ENC;
    //DAC_0 -> P6.6
    //DAC_1 -> P6.7
    //DAC reference Vref
    //12 bits resolution
    //Immediate load
    //DAC full scale output
    //Medium speed/
    //Straight binary
    //Not grouped
```

OA0 configuration

The OA module of the MSP430FG4168 has three operational amplifiers with wide utilization flexibility. For this laboratory it is set up using the OA0 in non-Inverting PGA mode with the following configuration:

- The inverting input is connected to the DAC12 channel 0;

- The amplifier gain is configured as unity;

- The input is configured in rail-to-rail mode;

- The output is connected to the channel A1.

```
OA0CTL1 |= OAFC_4 | OAFBR_0;
    //OAFBRx (feedback resistor) = 000b -> Tap 0
(G=1)
    //OAFCx (OAx function) = 100b -> Non-inverting
PGA
    //OARRIP = 0b -> OAx input range is rail-to-
rail

OA0CTL0 |= OAP_2 | OAPM_3 | OAADC1;
    //OANx (Inverting input) = XXb -> not
important
    //OAPx (Non-inverting input) = 10b -> DAC0
internal
    //OAPMx (Slew rate select) = 11b -> Fast
    //OAADC1 (OA output) = 1b -> output connected
to A1
    //OAADC0 (OA output) = 0b -> output not
connected A12
```

ADC12 ISR

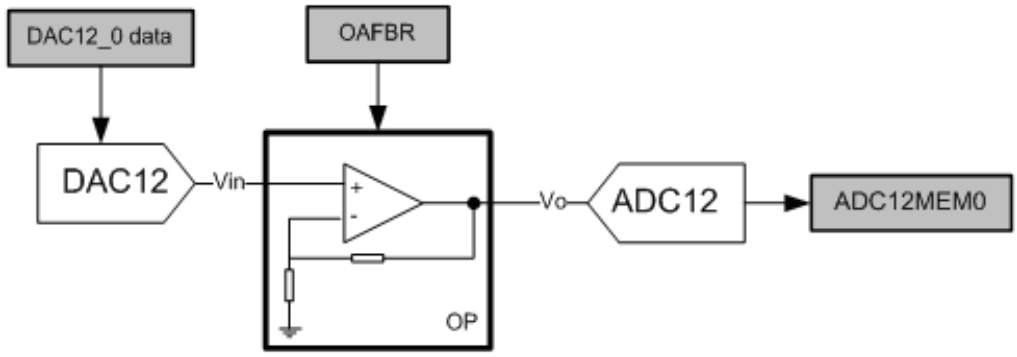
```
#pragma vector=ADC12_VECTOR
__interrupt void ADC_ISR(void)
{
 int x;
 x = ADC12MEM0; // Reads data
 ADC12CTL0 |= ADC12SC; // Start new conversion
}
```

Timer_A ISR

```
#pragma vector=TIMERA1_VECTOR
__interrupt void TimerA_ISR (void)
{
  ADC12CTL0 &= ~ADC12SC; //start new conversion
  TACTL &= ~TAIFG;
}
```

Analysis of operation

This laboratory uses the previous modules to construct an analogue signal chain as shown in Figure 1.
Analogue signal chain structure.



The input voltage V_{IN} is in the range 0 V and 2.5 V, with a resolution of:

$\Delta V_{IN} = (2.5 \times V_{REF}) / 2^{12} = 0.6$ mV

The V_{IN} value is controlled by the value in the DAC12_0DATA register.

The output voltage V_o has the same characteristics as the input voltage, but scaled by a multiplication factor (gain), attributed by the OA. The OA gain is selectable through the OAFBR field in the OA0CTL1 register.

The V_o conversion result is stored in the ADC12MEM0 register.

Once the signal chain modules are configured in accordance with the previous steps, initiate the experiment by completing the file, compiling it and running it on the Experimenter's board. For the evaluation of the peripherals discussed during this laboratory, set a breakpoint on the ADC12_ISR and perform the following operations:

- Configure the DAC12_0DATA register with the value 0xFF. With the aid of a voltmeter, measure the analogue input voltage A6 (DAC12 channel 0 output). The value should be in the region of 0.15 V;

- Measure the input voltage A1 (OA0's output). The voltage value should be the same;

- Execute the code. Verify the ADC12's conversion result. The value should be similar to the one of the DAC12_0DATA register;

- Double the amplifier gain (2x). Verify the voltage at A0. It should be the double of the input voltage A1 (OA0's output) given in step 2;

- Execute the code. Verify the ADC12's conversion result. The value should be two times the value of the DAC12_0DATA register;

- Execute further modifications in order to evaluate the digital-to-analogue and analogue-to-digital conversion. Do not exceed the Vo maximum value (2.5 V).

This example and many others are available on the MSP430 Teaching ROM.

Request this ROM, and our other Teaching Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Laboratory Signal Acquisition: Lab3 - SD16_A ADC conversion
Using the eZ430-F2013 Development Tool use SD16_A ADC to perform a single temperature sample on channel A6 (on-chip temperature sensor) each minute during 1 hour.

Laboratory Signal Acquisition: Lab3 - SD16_A ADC conversion

Introduction

This laboratory gives examples of the uses of the ADC types available in the hardware development kits. A different laboratory is developed for each kit, taking into account that both the ADC10 and the SD16_A laboratories implement a temperature data logger. The ADC12 laboratory also uses operational amplifiers to perform the analogue signal conditioning.

Overview

This laboratory (Lab3_ADC.c) implements a temperature data logger using the hardware kit's integrated temperature sensor. The device is configured to perform a data acquisition once every minute for one hour. Each temperature's (ºC) value is transferred to flash info memory segment B and C. When the microcontroller is not performing any task, it enters into low power mode.

Resources

The SD16_A module uses V_{REF+} = 1.2 V as reference voltage.

It is necessary to select the channel 6 of the SD16_A to use the integrated temperature sensor as an input. Timer_A generates an interrupt once every second, which starts conversion on the SD16_A. At the end of conversion,

an interrupt is requested by the converter and the temperature value is written to flash memory.

The voltage value is converted into temperature using the mathematical expression provided in the eZ430-F2013 data sheet. After transferring the value to the flash memory, the system returns to low power mode LPM3.

The resources used by the application are:

- SD16_A;

- Timer_A;

- Ports I/O;

- Interrupts;

- Low power mode.

Software application organization

The application starts by stopping the Watchdog Timer.

System tests for the presence of calibration constants in info memory segment A. The CPU execution will be trapped if it does not find this information.

The digital controller oscillator (DCO) is set to 1 MHz to provide clock sources for MCLK and SMCLK, while the Basic Clock System+ is configured to set ACLK to 1.5 kHz.

The controller’s flash timing is obtained from MCLK, divided by three to comply with the device specifications.

Port P1.0 is configured as output and will blink the LED once every second.

The SD16_A is configured to use the input channel corresponding to the on-chip temperature sensor (channel A6). The configuration includes the activation of the internal reference voltage: V_{REF+} = 1.2 V and the selection of SMCLK as clock signal. The converter is configured to perform a single conversion in bipolar mode and offset binary format. At the end of conversion an interrupt is requested.

The Timer_A is configured to generate an interrupt once every second. ACLK/8 is selected as the clock signal using VLOCLK as clock source and will count until it reaches the TACCR0 value (up mode). The system enters into low power mode and waits for an interrupt.

Flash memory pointers and interrupt counters are initialized. The Timer_A ISR increments variable counter and when this variable reaches the value 60 (1 minute), the software start of conversion is requested. At the end of this ISR, the system returns to low power mode.

When the SD16_A ends the conversion, an interrupt is requested. While variable min is lower than 60, the temperature is written in flash memory. The memory pointer is increased by two (word). When min = 60, the system stops operation.

System configuration

DCO configuration

Adjust the DCO frequency to 1 MHz by software using the calibrated DCOCTL and BCSCTL1 register settings stored in information memory segment A.

```
if (CALBC1_1MHZ == 0xFF || CALDCO_1MHZ == 0xFF)
    {
      while(1); // If calibration constants erased
                // do not load, trap CPU!!
    }
```

```
DCOCTL = CALDCO_1MHZ; // Set DCO to 1 MHz
```

Basic Clock module+ configuration

Set MCLK and SMCLK to 1 MHz. Use the internal very low power VLOCLK source clock to ACLK/8 clock signal as low frequency oscillator (12 kHz):

```
BCSCTL1 = DIVA_3; // ACLK = 1.5 kHz
BCSCTL3 = LFXT1S_2; // Set VLOCLK (12 kHz)
```

SD16_A configuration

The SD16_A's input channel is the integrated temperature sensor (A6) and it uses the signal V_{REF+} (1.2 V) as reference voltage. The SD16_A clock source is SMCLK. Configure the SD16_A to perform a single conversion and enable its interrupts. What are the values to write to the configuration registers?

```
SD16CTL = SD16REFON + SD16SSEL_1; // 1.2V ref,
SMCLK
SD16INCTL0 = SD16INCH_6; // Temp. sensor: A6+/-
SD16CCTL0 = SD16SNGL + SD16IE; // Single conv,
int. enable


//*************************************************
*********
// SD16_A Interrupt Service Routine
//*************************************************
*********
#pragma vector=SD16_VECTOR
__interrupt void SD16ISR(void)
```

```
 {
  unsigned int temperature;
  if (min <= 60)
   {
    temperature = (SD16MEM0-0x8000)/84 - 232;
    write_int_flash(memo_ptr,temperature);
    memo_ptr += 2;
   }
  else
   {
    _NOP();
   }
}
```

Timer_A configuration

Configure Timer_A register to enable an interrupt once every second. Use the ACLK clock signal as the clock source. This timer is configured in up mode in order to count until the TAR value reaches the TACCR0 value.

```
TACCTL0 = CCIE; // CCR0 interrupt enabled~
TACCR0 = 1500; // this count corresponds to 1 sec
TACTL = TASSEL_1 | MC_1 | ID_0; // ACLK, up mode
to CCR0


//**************************************************
*********
// Timer_A Interrupt Service Routine
//**************************************************
*********
#pragma vector=TIMERA0_VECTOR
__interrupt void TimerA0_ISR (void)
 {
  counter++;
  P1OUT ^= 0x01; // LED toogle
```

```
if (counter == 60)
    {
     min++;
     counter = 0;
     SD16CCTL0 |= SD16SC; // Start SD16 conversion
    }
}
```

Analysis of operation

Measure the temperature variation over 1 hour

After compiling the project and starting the debug session, before running the application, put a breakpoint at line of code with the `_NOP()` instruction. Go to breakpoint properties and set action to Write data to file. Name the file as Temp.dat and define the data format as integer. The data starts at address `0x01040` with a length of `3C`. Run the application and let the temperature data logger acquire the values over 1 hour. Use a heater or a fan to force temperature variations during the measurement period. When execution reaches the breakpoint, the file will be available in your file system. Construct a graph using Excel or a similar tool, to plot the temperature variation obtained by the data logger.

This example and many others are available on the MSP430 Teaching ROM.

Request this ROM, and our other Teaching Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Laboratory Signal Acquisition: Lab4 - Voltage signal comparison with Comparator_A
Using the MSP-EXP430FG4618 Development Tool analyse the Comparator_A operation. A voltage is applied to one of the Comparator's inputs, generated optionally by the DAC12 or by other external source. Whenever the external voltage value is higher than the comparison value internally generated, it is generated an interrupt that switches the LED state.

Laboratory Signal Acquisition: Lab4 - Voltage signal comparison with Comparator_A

Introduction

This laboratory gives examples of the uses of the ADC types available in the hardware development kits. A different laboratory is developed for each kit, taking into account that both the ADC10 and the SD16_A laboratories implement a temperature data logger. The ADC12 laboratory also uses operational amplifiers to perform the analogue signal conditioning.

Overview

This laboratory (Lab4_ADC.c) analyses Comparator_A operation. A voltage is applied to one of the Comparator's inputs, generated either by the DAC12 or by other external source. Whenever the external voltage value is higher than the comparison value internally generated, an interrupt is generated that switches the LED state.

Resources

The resources used by the application are:

- DAC12;

- Comparator_A;

- Digital IO;

- Timer_A.

Software application organization

The application starts by stopping the Watchdog Timer.

Timer_A is configured to generate an interrupt once every msec, and updates the DAC12 output in order to provide a ramp signal.

The Comparator_A's output is configured to be accessible at pin P6.6, which is available on Header 4 pin 7. The signal applied to CA0 input is compared with 0.5 V_{cc} internal reference voltage. Every time that a compare match occurs, an interrupt is requested and switches the state of LED1.

System configuration

Comparator_A configuration

Configure the registers in order to receive the external signal at the CA0 input and compare it with the internal reference 0.5 V_{cc}. Enable the comparator with an interrupt triggered on a low-to-high transition of the comparator output.

```
CACTL1 = CAON | CAREF_2 | CARSEL | CAIE;  //
Enable comp, ref = 0.5*Vcc

CACTL2 = P2CA0; // Pin to CA0

P2DIR |= 0x042; // P2.1 and P2.6 output direction
P2SEL |= 0x040; // P2.1 = LED1 and P2.6 = CAOUT
```

```
CACTL1 |= CAIE; // Setup interrupt for Comparator


//*************************************************
*********
// Comp_A interrupt service routine -- toggles LED
//*************************************************
*********
#pragma vector=COMPARATORA_VECTOR
__interrupt void Comp_A_ISR (void)
{
 CACTL1 ^= CAIES; // Toggles interrupt edge
 P2OUT ^= 0x02; // Toggle P2.1
}
```

ADC12 configuration

```
ADC12CTL0 = REF2_5V + REFON; // Internal 2.5V ref
on
```

DAC12 configuration

```
DAC12_0DAT = 0x00; // DAC_0 output 0V

DAC12_0CTL = DAC12IR | DAC12AMP_5 | DAC12ENC;
// DAC_0->P6.6
// DAC reference Vref
// 12 bits resolution
// Imediate load
// DAC full scale output
// Medium speed/current
// Straight binary
// Not grouped
```

Timer_A configuration

```
// Compare mode
TAR = 0; // TAR reset
TACCR0 = 13600; // Delay to allow Ref to settle
TACCTL0 |= CCIE; // Compare-mode interrupt
TACTL = TACLR + MC_1 + TASSEL_2; // up mode, SMCLK

// Interrupt enable
TAR = 0; // TAR reset
TACCTL0 = CCIE; // CCR0 interrupt enabled
TACCR0 = 32; // 1 msec counting period
TACTL = TASSEL_1 | MC_1 | ID_0; // ACLK, up mode


//*************************************************
*********
// ISR to TACCRO from Timer A
//*************************************************
*********
#pragma vector=TIMERA0_VECTOR
__interrupt void TimerA0_ISR (void)
{
 DAC12_0DAT++;

if (DAC12_0DAT == 0xFFF)
    DAC12_0DAT = 0;

    if (flag == 1) // if flag active exit LPM0
    {
     flag = 0;
     LPM0_EXIT;
    }
}
```

Analysis of operation

The experimental verification of this laboratory can be accomplished by connecting the DAC12's output, available on Header 8 pin 7, to the Comparator_A's input CA0, available on Header 4 pin 7.

Observe the signals wave form at the Comparator_A's input and output using an oscilloscope. The LED1 switches state whenever the input's voltage value is lower than the compare value.

This example and many others are available on the MSP430 Teaching ROM.

Request this ROM, and our other Teaching Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Laboratory DAC: Lab1 - Voltage ramp generator
Using the MSP-EXP430FG4618 Development Tool and the MSP430FG4618 device implement a voltage ramp generator Basic Timer1.

Laboratory DAC: Lab1 - Voltage ramp generator

Introduction

This laboratory gives an example of the use of the DAC available in the MSP-EXP430FG4618 Development Tool. The DAC module reference is obtained from the ADC module. The DAC is configured with 12 bits resolution in straight binary format. The DAC's output value is updated every 1 msec by a Timer_A ISR. The buttons SW1 and SW2 are used to manually modify the DAC's output.

Overview

This laboratory (Lab1_DAC.c)implements a voltage ramp generator. The DAC module reference is obtained from the ADC module. The DAC is configured with 12-bit resolution, in straight binary format. The output of the DAC value is updated once every 1 msec by an interrupt service routine (ISR) generated by Timer_A. The push buttons SW1 and SW2 are used to manually modify the output of the DAC value. When the microcontroller is not performing any task, it enters low power mode.

Resources

The DAC12_0 module uses V_{REF+} as reference voltage. It is therefore necessary to activate this reference voltage in the ADC12 module.

The DAC12_0 is connected to Port P6.6 on the Header 8 pin 7. Connect the oscilloscope probe to this port pin.

The output of the DAC is updated whenever Timer_A generates an interrupt. This peripheral is configured to generate an interrupt with a 1 msec time period.

After refreshing the output of the DAC, the system returns to low power mode LPM3.

The push buttons SW1 and SW2 allow the output of the DAC value to be changed manually.

The resources used by the application are:

- Timer_A;

- DAC12;

- I/O ports;

- FLL+;

- Interrupts.

Software application organization

The application starts by stopping the Watchdog Timer.

Then, the ADC12's reference voltage is activated and set to 2.5 V. A delay is used to allow the reference voltage to settle. During this time period, the device enters low power mode LPM0. The delay period, which is controlled by Timer_A, enables an interrupt when it completes. The interrupt wakes the device and proceeds with the execution of the application.

Timer_A is reconfigured to generate an interrupt once every 1 msec. This interrupt service routine (ISR) updates the output of the DAC.

Ports P1.0 and P1.1 are connected to buttons SW1 and SW2. The ports are configured as inputs with interrupt capability, such that the ISR can decode

which button is pushed. If the interrupt source is due to button SW1, then the output of the DAC is increased. If the interrupt source is due to button SW2, then the output of the DAC is decreased.

System configuration

FLL+ configuration

```
FLL_CTL0 |= DCOPLUS | XCAP18PF; // DCO+ set,
                                // freq = xtal x D
x N+1

SCFI0 |= FN_4; // x2 DCO freq,  // 8MHz nominal
DCO

SCFQCTL = 121;                  // (121+1) x 32768
x 2 = 7.99 MHz
```

Reference voltage selection

The DAC12_0 uses the signal V_{REF+} as reference voltage. What is the value to write to the configuration register in order to obtain the internally available reference?

```
ADC12CTL0 = REF2_5V | REFON; // Internal 2.5V ref
on
```

DAC12 configuration

The DAC12_0 is configured with 12-bit resolution. The output is updated immediately when a new DAC12 data value is written in straight binary

data format to the DAC12_0DAT register.

The full-scale output must be equal to the V_{REF+} 2.5 V internal reference voltage. Choose a compromise solution between the settling time and current consumption, by selecting a medium frequency and current for both input and output buffers. Configure the following register in order to meet these specifications:

```
DAC12_0DAT = 0x00; // DAC_0 output 0V

DAC12_0CTL = DAC12IR | DAC12AMP_5 | DAC12ENC;
    // DAC_0 -> P6.6,
    // DAC_1 -> P6.7,
    // DAC reference Vref,
    // 12 bits resolution,
    // Immediate load,
    // DAC full scale output,
    // Medium speed/current,
    // Straight binary,
    // Not grouped
```

Timer_A configuration

Configure Timer_A register to enable an interrupt once every 1 msec. Use the ACLK clock signal as the clock source. This timer is configured in count up mode in order to count until the TAR value reaches the TACCR0 value.

```
// Before entering in LPM0:
TACTL = TACLR | MC_1 | TASSEL_2; // up mode, SMCLK

// Timer_A ISR:
TAR = 0;                          // TAR reset
TACCR0 = 13600;                   // Delay to allow
Ref to settle
```

```
TACCTL0 |= CCIE;                    // Compare-mode
interrupt
TACTL = TACLR | MC_1 | TASSEL_2; // up mode, SMCLK


//*************************************************
*********
// ISR to TACCRO from Timer A
//*************************************************
*********
#pragma vector=TIMERA0_VECTOR
__interrupt void TimerA0_ISR (void)
{
 DAC12_0DAT++; // Increase DAC's output
 if (DAC12_0DAT == 0xfff)
 DAC12_0DAT = 0; // reset DAC's output
 if (flag == 1) // if flag active exite LPM0
 {
  flag = 0;
  LPM0_EXIT;
 }
}
```

I/O Ports configuration

Port P1 uses the bits P1.0 and P1.2 to activate the ISR whenever the push buttons SW1 and SW2 are activated (low-to-high transition).

```
// SW1 and SW2 ports configuration
P1SEL &= ~0x03; // P1.0 and P1.1 I/O ports
P1DIR &= ~0x03; // P1.0 and P1.1 digital inputs
P1IFG = 0x00;       // clear all interrupts
pending
P1IE |= 0x03;       // enable port interrupts
```

DAC12_0 is connected to P6.6. Configure P6 as a special function output:

```
// P6.6 (DAC12_0 output)
// There is no need to configure P6.6 as a
// special function output since it was configured
in the
// DAC12 configuration register (DAC12_0CTL) using
// DAC12OPS = 0

//*************************************************
*********
// Port1 Interrupt Service Rotine
//*************************************************
*********
#pragma vector=PORT1_VECTOR
__interrupt void PORT1_ISR (void)
{
 if (P1IFG & 0x01) // SW1 generate interrupt
 DAC12_0DAT += 400;    // DAC's output increases


if (P1IFG & 0x02) // SW2 generate interrupt
 DAC12_0DAT -= 400;   // DAC's output decreases


P1IFG = 0x00; // clean all pending interrupts
}
```

Analysis of operation

Observe the analogue signal using an oscilloscope

After compiling the project and starting the debug session, monitor the operation of the application using an oscilloscope probe connected to pin 7

of Header 8 (P6.6).

Measure the current drawn

Assign different values to the bits set in DAC12AMP0. Suspend the execution of the application then directly change the registers. Do not forget that this change requires suspending the operation of the DAC12 by disabling the bit DAC12ENC. Afterwards, this bit must be enabled.

Please note the special cases relating to:

- DAC12 off;

- High impedance output and DAC12 off;

- Output: 0 V.

This example and many others are available on the MSP430 Teaching ROM.

Request this ROM, and our other Teaching Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Laboratory DMA: Lab1 - Data Memory transfer triggered by software
Using the DMA controller included in the MSP-EXP430FG4618 Development Tool analysed the data transfer between two regions of memory.

Laboratory DMA: Lab1 - Data Memory transfer triggered by software

Introduction

This laboratory gives an example of the use of the DMA peripheral available in the MSP-EXP430FG4618 Development Tool. It requires the configuration of the DMA Source and Destination Addresses Registers, DMA Size Address Register; DMA Control Registers and DMA Channel 0 Control Register in order to transfer data between two regions of memory.

Overview

During this laboratory (Lab1_DMA.c), the data transfer between two regions of memory is analyzed. The order of transfer is controlled by software.

Resources

The following resource is used in this laboratory:

- DMA controller.

Software application organization

The software begins by disabling the watchdog timer. Port P2.1 is set as an output with a logic low level.

The memory addresses of the data vectors are passed to the source data address DMA0SA and destination address DMA0DA registers.

The number of words to be transferred is loaded in the DMA0SZ (size) register.

The DMA channel 0 is configured so that the data transfer trigger is controlled by software, in order that after each transfer, the source and destination addresses are correctly incremented.

The application enters an infinite loop, where port P2.1 state is switched just before initiating the data transfer.

System configuration

DMA channel configuration:

The source address and destination address of the data must be loaded into their respective registers:

```
DMA0SA = (void (*)( )) &tab_1; // Start block address

DMA0DA = (void (*)( )) &tab_2; // Destination block address
```

To move a total of 32 words, what is the value to write to the data size register?

```
DMA0SZ = 0x0020; // Block size
```

The DMA channel must be configured to transfer the word under software control. The source and destination addresses should be incremented immediately after each of the transfers.

```
DMA0CTL=DMADT_0 | DMASRCINCR_3 | DMADSTINCR_3 |
DMAEN;
    // Single transfer,
    // DMA source and destination addresses
increment,
    // Enable DMA0
```

Analysis of operation

In the Memory window, the addresses of data vector Tab_1 and Tab_2 addresses are displayed. The contents of these blocks must be identified in memory.

Add a breakpoint at line of code that performs the switching of port P2.1 state.

Execute the application, and whenever the breakpoint is reached, the execution of the application will be suspended. Observe the data being gradually transferred from source to destination.

The data transfer is suspended once the 32 elements of the source data vector have been transferred.

This example and many others are available on the MSP430 Teaching ROM.

Request this ROM, and our other Teaching Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Laboratory DMA: Lab2 - Sinusoidal waveform generator
Using the DMA controller included in the MSP-EXP430FG4618 Development Tool transfer a sinusoidal wave to the DAC12_0's output. Timer_A operates in upmode with TACCR1 loading DAC12_0 on rising edge and DAC12_0IFG triggering next DMA1 transfer. DAC12_0 uses internal 1.5V reference.

Laboratory DMA: Lab2 - Sinusoidal waveform generator

Introduction

This laboratory gives an example of the use of the DMA peripheral available in the MSP-EXP430FG4618 Development Tool. It requires the configuration of the DMA Source and Destination Addresses Registers, DMA Size Address Register; DMA Control Registers, DMA Channel 0 Control Register, DAC12 control register and Timer_A control register in order to generate a sinusoidal waveform.

Overview

This laboratory (Lab2_DMA.c) uses the DMA controller to automatically transfer data between data memory and the DAC12 data register. A sinusoidal waveform is produced at the output of the DAC, without CPU intervention.

Resources

This laboratory uses the following peripherals:

- DMA controller;

- DAC;

- ADC (reference generator: V_{REF+});

- Timer_A;

- Low power mode.

Software application organization

The successive samples needed to produce the sinusoidal waveform using the DAC are stored in the data vector **Sin_tab**, which contains 32 points:

```
//-------------------------------------------------
---------
// 12-bit Sine Lookup table with 32 steps
//-------------------------------------------------
---------
int Sin_tab[32] = {2048, 2447, 2831, 3185, 3495,
3750, 3939, 4056,
                   4095, 4056, 3939, 3750, 3495,
3185, 2831, 2447,
                   2048, 1648, 1264,  910,  600,
345,  156,   39,
                      0,   39,  156,  345,  600,
910, 1264, 1648};
```

The software begins by disabling the watchdog timer, followed by activating the internal reference voltage V_{REF+}. The source and destination registers of the data vector to be transferred by the DMA channel are loaded into the data vector Sin_tab (source) address and with the DAC12 data register (destination) address. There are 32 data values to be transferred.

The data transfer is initiated whenever the DAC12IFG flag is enabled. In this application, the DAC interrupt should be disabled.

The DMA controller is configured to operate in repeat mode, to transfer a word whenever the previous event occurs. The data source address is set to

increment after each transfer, while the destination address must remain constant.

The timer is set to generate the PWM signal through the capture/compare unit TACCR1. SMCLK is the clock signal that counts up to the value in the TACCR0 register.

Finally, the settings and interrupts are enabled and the device enters into low power mode LPM0.

System configuration

DAC12 reference voltage activation

The DAC12 requires a reference voltage. One of the options is to use the internal voltage V_{REF+}. Set the ADC12CTLO register to activate this voltage:

```
ADC12CTL0 = REFON; // Internal reference
```

DMA Controller configuration:

Configure the registers DMA0SA (source), DMA0DA (destination) and DMA0SZ (size) to transfer 32 words between the source vector Sin_tab and the DAC12_0DAT data destination register:

```
DMA0SA = (void (*)( ))&Sin_tab;    // Source block
address

DMA0DA = (void (*)( ))&DAC12_0DAT; // Destination
single address

DMA0SZ = 0x20;                          // Block
```

```
size
```

Configure the register DMACTL0 to provide a data transfer whenever the DAC12IFG flag is set:

```
DMACTL0 = DMA0TSEL_5; // DAC12IFG trigger
```

Configure the register DMA0CTL to carry out a repeated simple data transfer, increasing the data source address:

```
DMA0CTL = DMADT_4 | DMASRCINCR_3 | DMAEN;
    // Repeated single transfer,
    // DMA source address increment,
    // since DMASRCBYTE = 0, the source address
increments by
    // two (word-word)
```

Setup DAC12

The DAC12 will update its output whenever there is the activation of the signal TA1. The DAC full-scale should be 1x reference voltage. Choose a medium relationship between the DAC's current and average conversion speed:

```
DAC12_0CTL = DAC12LSEL_2 | DAC12IR | DAC12AMP_5 |
DAC12IFG | DAC12ENC;
    // Rising edge of Timer_A.OUT1 (TA1),
    // DAC12 full-scale output: 1x reference
voltage,
    // Input and output buffers: Medium
freq./current,
    // Enable DAC12
```

Timer_A configuration

Timer_A is responsible for synchronizing data transfers between memory and the DAC12. The Timer_A input receives as the SMCLK signal (1.048576 MHz) and must have a 30 msec counting period. What value needs to be written to TACCR0, in order to achieve this counting period:

```
TACCR0 = 32-1;           // Clock period of TACCR0
TACTL = TASSEL_2 | MC_1; // SMCLK, continuous mode
```

The capture/compare unit TACCR1 should generate a PWM signal in set/reset mode. Configure the unit appropriately:

```
TACCTL1 = OUTMOD_3; // TACCR1 set/reset
TACCR1 = 20;        // TACCR1 PWM Duty Cycle
```

Analysis of operation

The verification of this laboratory is achieved by using an oscilloscope probe to monitor the output of the DAC12 Channel 0, available on header 8 pin 6.

This example and many others are available on the MSP430 Teaching ROM.

Request this ROM, and our other Teaching Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Laboratory Hardware Multiplier: Lab1 - Multiplication without hardware multiplier
Using the MSP-EXP430FG4618 Development Tool and the MSP430FG4618 device analyse the MSP430's performance when makes a multiplication operation without the hardware multiplier. The execution time is measured with an oscilloscope.

Laboratory Hardware Multiplier: Lab1 - Multiplication without hardware multiplier

Introduction

This laboratory explores the hardware multiplier peripheral. It is composed of three different tasks, each of which evaluates a different characteristic of the hardware multiplier peripheral:

- Multiplication operation execution time, with and without the hardware multiplier.

-Differences between the use of the operator "*" and direct write to the hardware multiplier registers.

- Task operational analysis, in which the active power and the RMS value of an electrical system are calculated.

Overview

This laboratory explores and analyses the MSP430's performance when it performs multiply operations without the hardware multiplier. The execution time is measured using an oscilloscope.

Resources

This laboratory only uses Port P2.1 connected to LED2 in order to measure the execution time of the multiply operation when it is performed by a software routine.

The default configuration of the FLL+ is used. All the clock signals required for the operation of the components of this device take their default values.

Software application organization

- The application starts by stopping the Watchdog Timer;

- Port P2.1 is configured as an output with the pin at a low level;

- The variables a and b to be multiplied are initialized;

- The multiplication of the two variables is performed between toggle P2.1 instructions;

- This application ends by putting the device into low power mode LPM4.

System configuration

Go to **Properties > TI Debug Settings** and select the **Target** tab. Uncheck the **automatically step over functions without debug information when source stepping** in order to allow stepping into the multiply routine;

Go to **Properties > C/C++ Build > Linker MSP430 Linker v3.0 > General options** and choose the option **None** at the **Link in hardware version of RTS mpy routine**. With this linker option, the application (Lab1_HM.c) will be built without the hardware multiplier and all multiplication operations will be performed by the software routine.

Rebuild the project and download it to the target.

Analysis of operation

Software multiplication routine analysis

- Connect the oscilloscope probe to port P2.1 available on Header 4 pin 2;

- Put the cursor at line of code 51 {c = a*b} and **Run to line**;

- Go to **Disassembly** view and switch to **mixed disassembly view** in order to show both C and Assembly code;

- Observe that the variables **a** and **b** are passed by registers and the `#__mpyi` routine is called;

- Run the code step-by-step with the **Disassembly** view active. This action will lead to the software multiply routine;

- As the software multiply routine source code is not available, switch to **Assembly** view only;

- Run the application step-by-step until the `RETA` instruction;

- This multiplication is a time-consuming CPU operation.

Measurement of the multiply operation execution time

- Restart the application. It will run from the beginning;

- Put the cursor on line of code 56 `{_BIS_SR(LPM4)}` and **Run to line**;

- Measure the time pulse time width using the oscilloscope;

- This software multiply operation takes around 54 µsec.

This example and many others are available on the MSP430 Teaching ROM.

Request this ROM, and our other Teaching Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Laboratory Hardware Multiplier: Lab2 - Multiplication with hardware multiplier
Using the MSP-EXP430FG4618 Development Tool and the MSP430FG4618 device analyse the MSP430's performance when makes a multiplication operation with the hardware multiplier. Two different situations are analysed: - Using the "*" operator; - Accessing the hardware multiplier registers directly. The execution time is measured with an oscilloscope.

Laboratory Hardware Multiplier: Lab2 - Multiplication with hardware multiplier

Introduction

This laboratory explores the hardware multiplier peripheral. It is composed of three different tasks, each of which evaluates a different characteristic of the hardware multiplier peripheral:

- Multiplication operation execution time, with and without the hardware multiplier.

-Differences between the use of the operator "*" and direct write to the hardware multiplier registers.

- Task operational analysis, in which the active power and the RMS value of an electrical system are calculated.

Overview

This laboratory explores and analyses the MSP430's performance when it performs multiply operations using the hardware multiplier peripheral. Two different variants are analysed:

- Using the "*" operator;

-Accessing the hardware multiplier registers directly.

The execution times are measured with an oscilloscope.

Resources

This laboratory only uses Port P2.1 connected to LED2 in order to measure the execution time of the multiplication operation, when it is performed by the hardware multiplier.

The default configuration of the FLL+ is used. All the clock signals required for the operation of the components of the device take their default values.

Software application organization

The application begins by stopping the Watchdog Timer;

Port P2.1 is configured as an output with the pin at a low level;

The code can be broken down into two parts:

- In the first part of the code, the multiplication is performed with the “*” operator. This task is performed between P2.1 toggles, in order to determine the time required to perform this operation;

- The remaining part of the code is separated by some `_NOP()` operations. This coding allows analysis of the execution time using an oscilloscope. Here, the multiplication operation is performed by directly accessing the hardware multiplier registers. The multiplication of the variables is performed between toggle P2.1 instructions;

This application ends with the device entering low power mode LPM4.

System configuration

Go to **Properties > TI Debug Settings** and select the **Target** tab. Uncheck the **automatically step over functions without debug information when source stepping** in order to allow stepping into the multiply routine;

Go to **Properties > C/C++ Build > Linker MSP430 Linker v3.0 > General options** and choose the option **16 (default)** at the **Link in hardware version of RTS mpy routine**. With this linker option, the application (Lab2_HM.c) will be built with the 16-bit hardware multiplier peripheral contained in the Experimenter's board.

Rebuild the project and download to the target.

Analysis of operation

Analysis of hardware multiply routine with the "*" operator

- Connect the oscilloscope probe to port P2.1, which is connected to Header 4 pin 2;

- Put the cursor at line of code 55 {`c = a*b`} and **Run to line**;

- Go to **Disassembly** view and switch to **mixed disassembly** view in order to show both C and Assembly code;

- Observe that the variables **a** and **b** are passed to registers and `#__mpyi_hw` routine is called;

- Run the code step-by-step with the **Disassembly** view active. This action will lead to the multiply operation being performed by the hardware multiplier;

- As the hardware multiply routine source code is not available, switch to **Assembly** view only;

- The routine starts by pushing the Status Register onto the system stack (`PUSH` instruction) and disabling the interrupts (this always occurs when using the hardware multiplier peripheral);

- The next line of code exchanges data with the hardware multiplier;

- Then the SR is popped (`POP` instruction) from the system stack, restoring the system environment (data interrupt state restored);

- The routine finishes with a `RETA` instruction.

Analysis of hardware multiply operation with direct registers access

- Switch to the **C** view;

- Put the cursor at line of code 72 {`MPY = a`} and **Run to line**;

- The routine call operation is avoided, as shown in the **Disassembly** view. This exemplifies an energy saving procedure because it shows how less CPU clock cycles can be used.

Measurement of execution time of the multiply operation

- Restart the application. It will run from the beginning;

- Put the cursor at line of code 77 {`_BIS_SR(LPM4)`} and **Run to line**;

- Measure the pulse widths using the oscilloscope;

- The first time pulse corresponds to the hardware multiply routine with the operator “*”, and has a width of 42 µsec;

- The second time pulse corresponds to the hardware multiply register operation and has a width of 19 µsec;

- Comparing both time pulses and the time pulse obtained in *Lab1: Multiplication without the hardware multiplier*, it can be seen that with the hardware multiplier there is a significant reduction of the time required to perform a multiply operation;

- The smaller time pulse corresponds to the hardware multiply operation writing directly to the hardware multiplier registers. This reduction in time means less power consumption, which is very useful for the design of low-power applications.

This example and many others are available on the MSP430 Teaching ROM.

Request this ROM, and our other Teaching Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Laboratory Hardware Multiplier: Lab3 - RMS and active power calculation Using the MSP-EXP430FG4618 Development Tool and the MSP430FG4618 device analyse the MSP430's performance when makes a multiplication operation with the hardware multiplier. It is performed a task operational analysis in which is calculated the active power and the RMS value of an electrical system. The execution time is measured with an oscilloscope.

Laboratory Hardware Multiplier: Lab3 - RMS and active power calculation

Introduction

This laboratory explores the hardware multiplier peripheral. It is composed of three different tasks, each of which evaluates a different characteristic of the hardware multiplier peripheral:

- Multiplication operation execution time, with and without the hardware multiplier.

-Differences between the use of the operator "*" and direct write to the hardware multiplier registers.

- Task operational analysis, in which the active power and the RMS value of an electrical system are calculated.

Overview

This laboratory explores and analyses the MSP430 performance when it makes multiply operations using the hardware multiplier peripheral. In this laboratory, the active power and the RMS value of an electrical signal are calculated.

The execution times are measured using an oscilloscope.

Resources

This laboratory only uses Port P2.1 connected to LED2, in order to measure the execution time of the multiply operation when it is performed by the hardware multiplier.

The application uses the default configuration of the FLL+. All the clock signals required for the operation of the components of the device take their default values.

Software application organization

- The application starts by stopping the Watchdog Timer;

- Two `_NOP()` instructions are provided to associate breakpoints, in order to read current and voltage samples (`N = 200`) from files;

- Power is computed by applying the following formula:
Equation:

$$P \quad \frac{}{N}_{k}^{N} u_k i_k$$

- A signed multiply operation is performed by writing the first sample of current to **MPYS** and the first sample of voltage to **OP2**;

- The result of the multiplication is stored in the **RESHI** and **RESLO** registers;

- A loop is performed with a signed multiply and accumulate (**MACS**) operation;

- The final result is transferred from the **RESHI** and **RESLO** registers to the long variable **result**;

- The power is computed by dividing the variable **result** by the number of samples (N);

- Port P2.1 is active between **MACS** operations;

- The RMS current and voltage values are calculated from the following expressions:

Equation:

$$I \qquad \frac{\sum_{k}^{N} i_k i_k}{N}$$

Equation:

$$U \qquad \frac{\sum_{k}^{N} u_k u_k}{N}$$

- The two procedures are similar, with the exception of the square root (`sqrt`) operations;

- P2.1 is active during for all the RMS current calculation;

- The computation times of the sqrt and division operations are determined when the RMS voltage value is calculated;

- This application ends by putting the device into low power mode LPM4.

System configuration

Go to **Properties** > **TI Debug Settings** and select the **Target** tab. Uncheck the **automatically step over functions without debug information when source stepping** in order to allow stepping into the multiply routine;

Go to **Properties > C/C++ Build > Linker MSP430 Linker v3.0 > General options** and choose the option **16 (default)** at the **Link in hardware version of RTS mpy routine**. With this linker option, the application (Lab3_HM.c) will be built with the 16-bit hardware multiplier peripheral contained in the Experimenter's board.

Rebuild the project and download to the target.

Analysis of operation

Loading samples from files

- Insert a breakpoint at line of code 61 (first `_NOP()` operation);

- Edit **Breakpoint Properties** and choose the **Read Data from file** action;

- Configure the following data fields:

File: i.txt

Wrap around: `True`

Start address: `&i`

Length: `200`

- Include a breakpoint at line of code 63 (second `_NOP()` operation);

- Edit **Breakpoint Properties** and choose the **Read Data from file** action;

- Configure the following data fields:

File: u.txt

Wrap around: **True**

Start address: **&u**

Length: **200**

Computing active power

- Connect the oscilloscope probe to port P2.1, which is available at Header 4 pin 2;

- Put the cursor at the line of code 88 and **Run to line**;

- In the **Variables** view, add the global variable **P** and format it to decimal;

- The active power is in the region of 1204 W;

- The pulse width, as viewed on the oscilloscope, corresponds to the time to perform the 200 signed multiply and accumulate operations and is 5.4 msec.

Compute RMS current value

- Starting at the last step of the previous task, put the cursor at line of code 105 {`MPYS = u[0]`} and Run to line;

- Add the global variable **I** (RMS voltage);

- Set the value to 10;

- The pulse width, as viewed on the oscilloscope, corresponds to the time required to perform the 200 signed multiply and accumulate operations, 1 division operation and 1 square root operation, and is 12.6 msec;

Compute RMS voltage value

- Starting at the last step of the previous task, put the cursor at line of code 121 {`_BIS_SR(LPM4)`} and **Run to line**;

- Add the global variable **U** (RMS voltage);

- Set the value to 240;

- The pulse width, as viewed on the oscilloscope, corresponds to the time to perform the 200 signed multiply and accumulate operations, and is 6.8 msec;

This example and many others are available on the MSP430 Teaching ROM.

Request this ROM, and our other Teaching Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Laboratory Flash memory: Lab1 - Flash memory programming with the CPU executing the code from flash memory
Using the MSP-EXP430FG4618 Development Tool implement a flash memory programming with the CPU executing the code in the flash memory.

Laboratory Flash memory: Lab1 - Flash memory programming with the CPU executing the code from flash memory

Introduction

The TI MSP430 has an internal flash memory that can be used for data storage. Two different methods of writing to the flash memory are studied in this laboratory. The first method requires the CPU execution of the code resident in flash memory. The consequences of this procedure are discussed. In the second part of the laboratory, the flash write and erase operations are conducted with the CPU executing the code resident in RAM. The important details are highlighted.

Overview

This laboratory programs the internal flash memory with the CPU executing the code resident in flash memory. It requires to configure: - Flash memory controller; - Segment erase routine; and the - Flash write routine. The execution time of the different operations can be obtained with an oscilloscope connected on pin 2 of the Header 4 or analyzing the state of the LED (digital output P2.1).

Resources

This laboratory uses the flash memory controller. The operation of this device is monitored using a digital output port (P2.1).

The project must be compiled using the files (Lab1_Flash.c)and the command file lnk_msp430fg4618.cmd.

The code is resident in the flash memory, so whenever a flash write or erase operation occurs, the CPU access to this memory is automatically inhibited.

Software application organization

The software begins by disabling the Watchdog Timer. Then, port P2.1 is set as an output with a logic low level.

The flash memory controller is configured with the clock MCLK divided by 3. Thus the fFTG operating frequency lies within the specified limits of 257 kHz to 476 kHz.

A set of routines are provided to erase, write and copy the contents of a segment. The main tasks related to the flash memory handling are presented using this set of routines.

The information Segments A and B are erased first. Then, bytes are written to SegmentA and words are written to SegmentB. The contents of the information memory SegmentA are copied to the information SegmentB, overwriting the previous contents.

System configuration

Flash memory controller configuration

Configure the register FCTL2 to use clock MCLK divided by 3. Do not forget to enter the password to access the register.

```
FCTL2 = FWKEY | FSSEL0 | FN1; // MCLK/3 for Flash Timing Generator
```

Segment erase routine

Configure the registers FCTL1 and FCTL3 in order to initiate the flash segment erase process by writing an address belonging to the segment to be erased. Be sure to include the password to access the register.

```
FCTL1 = FWKEY | ERASE; // Set Erase bit
FCTL3 = FWKEY;         // Clear Lock bit
```

Block flash write and erase operations are carried out after erasing the segment:

```
//Flash block write and erase operations after
erasing the segment:
FCTL3 = FWKEY | LOCK; // Set LOCK bit
```

Flash write routine

Configure the registers in order to start writing to the flash memory. Be sure to include the password to access the register.

```
FCTL1 = FWKEY | ERASE; // Set Erase bit
FCTL3 = FWKEY;         // Clear Lock bit
```

Configure flash block write and erase operations and disable the write bit:

```
// Flash block write and erase operations and
disable the write bit
// after the writing process to the segment:
FCTL3 = FWKEY | LOCK; // Set LOCK bit
```

Analysis of operation

Execution time for the information segments erase operation

Put the cursor at line of code 124, located just after the second port P2.1 switching state. Execute the software until the cursor position is reached. The erase operation timing can be seen on an oscilloscope with the probe connected to pin 2 of the Header 4.

Bytes write in the information memory A

The routine `write_char_flash` allows writing a byte to flash memory. It receives the memory address where the byte should be stored.

Open the **memory** window, and add the address of the information memory A. The content of this address becomes visible after ordering its **rendering**. As we are writing a byte to flash, we must change the presentation of the memory contents. Choose the option **Column Size 1**, from the context menu of the **memory** window, through the option **Format**.

Now, during the execution of the for loop, the flash contents is written sequentially.

Bytes written in the information B memory

This routine is similar to the previous one. Note that now the flash write address is increased by two because a word occupies two bytes of memory.

The information is more readily observed when the memory contents display mode is restored to its initial state. Reset the default conditions in the option **Format** of the context menu.

Copy the contents of the information A memory to information B memory

The output port P2.1 is enabled before the copy process begins. The copy routine receives the start address of the source information segment and the start address of the destination information segment. The information is then successively read and written from one segment to another.

Port P2.1 is disabled at the end of the copy process. Thus, the task execution time can be measured using an oscilloscope.

This example and many others are available on the MSP430 Teaching ROM.

Request this ROM, and our other Teaching Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Laboratory Flash memory: Lab2 - Flash memory programming with the CPU executing the code in RAM
Using the MSP-EXP430FG4618 Development Tool implement a flash memory programming with the CPU executing the code in RAM.

Laboratory Flash memory: Lab2 - Flash memory programming with the CPU executing the code in RAM

Introduction

The TI MSP430 has an internal flash memory that can be used for data storage. Two different methods of writing to the flash memory are studied in this laboratory. The first method requires the CPU execution of the code resident in flash memory. The consequences of this procedure are discussed. In the second part of the laboratory, the flash write and erase operations are conducted with the CPU executing the code resident in RAM. The important details are highlighted.

Overview

This laboratory programs the internal flash memory with the CPU executing the code in RAM. It requires to configure: - Several flash storage management routines; - Check the state of the flag Wait; an the - Flash write routine. This procedure requires special attention during the project construction. The application begins copying the routines from flash to RAM. Directive MEMORY: Device's memory configuration. Identifies the memory ranges that are physically present on the device Directive SECTIONS: controls how the sections are built and reserved.

The execution time of the different operations can be obtained with an oscilloscope connected on pin 2 of the Header 4 or analyzing the state of the LED (digital output P2.1).

Resources

The tasks developed in the *Lab1: Flash memory programming with the CPU executing the code from flash memory* are executed again during this laboratory. The difference this time is that the software runs from RAM.

This process requires special procedures. The routines to run from RAM must be identified. The application must begin by copying the routines from flash to RAM.

The directive MEMORY determines the device's memory configuration. The memory can be organized in accordance with the system needs. This directive identifies the memory ranges that are physically present on the device. Each of these ranges has a set of features, such as:

- Name;

- Initial address;

- Length;

- Optional attributes set;

- Optional filling specifications.

The directive **Memory** is organized as described below.

```
MEMORY
    {
    name 1 [(attr)] : origin = constant, length =
constant [, fill = constant]

    name n [(attr)] : origin = constant, length =
constant [, fill = constant]
    }
```

The directive **SECTIONS** controls how the sections are built and reserved. The directive performs the following:

- Describes how the input sections are related to the output sections;

- Defines the output sections in the executable program;

- Defines where the output sections are placed in memory;

- Allows changing the name of the output sections;

The directive **SECTIONS** is organized as described below.

```
SECTIONS
    {
    name : [property [, property] [, property] . .
. ]
    name : [property [, property] [, property] . .
. ]
    name : [property [, property] [, property] . .
. ]
    }

    The following directives are possible:
    // Reserve memory space to load the section:
    Syntax: load = allocation or
    Allocation or
    > allocation

    // Define the memory space where the code
belonging to the section will run:
    Syntax: run = allocation or
    run > allocation
```

In this project, we intend to write the code to the flash memory, but we want it to be executed from RAM. The Linker offers a very simple way to accomplish this task. A memory space where the code is stored is associated with another memory space where it will run. The application transfers the code to the memory space, where it will be executed.

The memory spaces needed to store the routines are defined in the lnk_msp430fg4618_RAM.cmd file.

```
RAM_MEM : origin = 0x1100, length = 0x0200
FLASH_MEM : origin = 0x3100, length = 0x0200
```

The following sections are also defined:

```
.FLASHCODE : load = FLASH_MEM, run = RAM_MEM
.RAMCODE : load = FLASH_MEM
```

Software application organization

The software for this laboratory has the same structure as the *Lab1: Flash memory programming with the CPU executing the code from flash memory*.

The directive `#pragma CODE_SECTION (symbol, "section name")` reserves space for the "`symbol`" in a section called "`section name`". Thus, the routines are stored in the section "`.FLASHCODE`".

The routine `copy_flash_to_RAM` runs from the beginning of the program. It is responsible for transferring the flash contents to RAM.

The files (Lab2_Flash.c) and lnk_msp430fg4618_RAM.cmd must be included during the compilation.

Now, the code is executed from RAM. Check, whenever appropriate, the **Wait** bit state of the register FCTL3.

System configuration

Flash storage management routines

To store the flash management routines in the section ".FLASHCODE" complete the empty spaces:

```
#pragma CODE_SECTION(erase_segment,".FLASHCODE")
void erase_segment(int address)

#pragma
CODE_SECTION(write_char_flash,".FLASHCODE")
void write_char_flash(int address, char value)

#pragma CODE_SECTION(write_int_flash,".FLASHCODE")
void write_int_flash(int address, int value)

#pragma CODE_SECTION(copy_seg_flash,".FLASHCODE")
void copy_seg_flash(int address_source, int
address_destination)
```

Check the flag wait

At software key points, and whenever writing or erasing the flash memory, perform a delay before proceeding with the data writes. Complete the following line of code in order to suspend the program flow while the busy flag is not active.

```
while(FCTL3&BUSY); // Check BUSY flag
```

Analysis of operation

Analyse the differences between the different versions of the routines. Note that successive delays are placed in the versions to be executed from RAM.

This example and many others are available on the MSP430 Teaching ROM.

Request this ROM, and our other Teaching Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Laboratory Communications: Lab1 - Echo test using the UART mode of the USCI module
Using the MSP-EXP430FG4618 Development Tool and the MSP430FG4618 device use the USCI module in UART mode connected to a PC's I/O console. When the connection is established, the characters sequence writed at the console through the keyboard will be visualized on the console.

Laboratory Communications: Lab1 - Echo test using the UART mode of the USCI module

Introduction

The MSP430 contains built-in features for both parallel and serial data communication. This chapter describes the operation of these peripherals, and discusses the protocols, data formats and specific techniques for each type of data communication.

The communication modules available for the MSP430 family of microcontrollers are USART (Universal Synchronous/Asynchronous Receiver/Transmitter), USCI (Universal Serial Communication Interface) and USI (Universal Serial Interface). These provide asynchronous data transmission between the MSP430 and other peripheral devices when configured in UART mode. They also support data transmission synchronized to a clock signal through a serial I/O port in Serial Peripheral Interface (SPI) and Inter Integrated Circuit (I2C) modes.

Overview

This laboratory explores the USCI module in UART mode that will be connected to a Code Composer Essentials (CCE) IO console. When the connection is established, the character sequence written on the keyboard to the console will be displayed again on the console.

Resources

This laboratory uses the USCI module in asynchronous mode. The RX interrupt activates the service routine that reads the incoming character and sends it out again to the PC (computer), allowing the instantaneous display (echo) of the written character.

The resources used are:

- USCI module;

- Interrupts;

- IO ports:

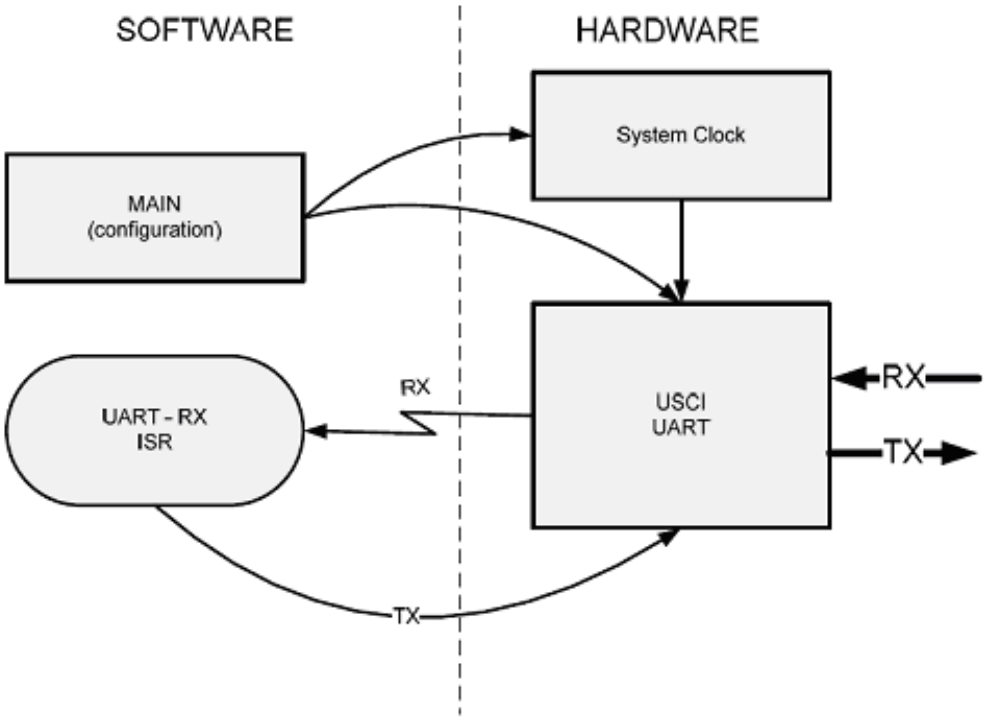
- System clock.

With the objective of allowing the generation of two different baud rates, a function has been added that configures the FLL+ and selects the base frequency for the UART. In this example it will be 8 MHz.

Software application organization

The proposed application (Lab1_Comm.c) is organized as shown in Figure 1. The main routine performs the necessary hardware configuration. Then, the hardware takes command of the software through the interrupt service routine generated by the reception of a new character.

The initial configuration sets the system clock to a frequency of 8 MHz.
Software application organization



System configuration

UART configuration

The connection will operate in the following mode:

- Parity disabled;

- LSB first;

- 8-bit data;

- One stop bit.

The module will operate in the following mode:

- Asynchronous;

- SMCLK source clock;

- No Receive erroneous-character interrupt-enable;

- No Receive break character interrupt-enable.

Based on these characteristics the following control registers are configured:

```
UCA0CTL0 = 0x00;
    // UCA0CTL0 =

//UCPEN|UCPAR|UCMSB|UC7BIT|UCSPB|UCMODEx|UCSYNC|
    //UCPEN (Parity) = 0b -> Parity disabled
    //UCPAR (Parity select) = 0b -> Odd parity
    //UCMSB (MSB first select) = 0b -> LSB first
    //UC7BIT (Character length) = 0b -> 8-bit data
    //UCSPB (Stop bit select) = 0b -> One stop bit
    //UCMODEx (USCI mode) = 00b -> UART Mode
    //UCSYNC = 0b -> Asynchronous mode

UCA0CTL1 = 0x81;
    // UCA0CTL1 =

//UCSSELx|UCRXEIE|UCBRKIE|UCDORM|UCTXADDR|UCTXBRK|
UCSWRST|
    //UCSSELx (USCI clock source select) = 10b ->
SMCLK
    //UCRXEIE = 0b -> Erroneous characters
rejected
    //UCBRKIE = 0b -> Received break characters
set
    //UCDORM = 0b -> Not dormant
    //UCTXADDR = 0b -> Next frame transmitted is
data
    //UCTXBRK = 0b -> Next frame transmitted is no
break
```

```
    //UCSWRST = 1b -> normally Set by a PUC
```

Baud rate generation

The module has an 8 MHz clock source and the objective is to establish a connection at 9600 Baud. It is necessary to select the baud rate generation in oversampling mode:

```
UCA0BR0 = 0x34;
UCA0BR1 = 0x00;
    //Prescaler = 8MHz/(16 x 9600) = 52 = 0x34
    //9600 from 8MHz -> SMCLK

UCA0MCTL = 0x11;
    // UCA0MCTL = UCBRFx | UCBRSx | UCOS16
    //UCBRFx (1st modulation stage) = 0001b ->
Table 19-4
    //UCBRSx (2nd modulation stage) = 000b ->
Table 19-4
    //UCOS16 (Oversampling mode) = 1b -> Enabled
```

Port configuration

In order to set the external interfaces at the USCI module, it is necessary to configure the I/O ports. Select the USCI peripheral in UART mode following the connections provided on the Experimenter's board:

```
P2SEL |= 0x30; // P2.4,P2.5 = USCI_A0 TXD,RXD
```

RX interrupt enable

To finish the module configuration, it is necessary to enable the receive interrupts:

```
IE2 |= UCA0RXIE; // Enable USCI_A0 RX interrupt
```

Analysis of operation

Once the USCI module is configured in accordance with the previous steps, compile it and run it on the Experimenter's board.

For the correct operation, there must be a connection between the Experimenter's board and the PC. If the CCE console is disabled, go to **Window > Show View > Console** to enable it. If necessary, configure the CCE console options in accordance to the connection details.

Once the program code is running, any character key pressed in the PC keyboard will be displayed on the CCE console.

This example and many others are available on the MSP430 Teaching ROM.

Request this ROM, and our other Teaching Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Laboratory Communications: Lab2 - Echo test using SPI
Using the MSP-EXP430FG4618 Development Tool establish a data exchange between the MSP430FG4618 and MSP430F2013 devices using the SPI mode. The MSP430FG4618 uses the USCI module while the MSP430F2013 uses the USI module.

Laboratory Communications: Lab2 - Echo test using SPI

Introduction

The MSP430 contains built-in features for both parallel and serial data communication. This chapter describes the operation of these peripherals, and discusses the protocols, data formats and specific techniques for each type of data communication.

The communication modules available for the MSP430 family of microcontrollers are USART (Universal Synchronous/Asynchronous Receiver/Transmitter), USCI (Universal Serial Communication Interface) and USI (Universal Serial Interface). These provide asynchronous data transmission between the MSP430 and other peripheral devices when configured in UART mode. They also support data transmission synchronized to a clock signal through a serial I/O port in Serial Peripheral Interface (SPI) and Inter Integrated Circuit (I2C) modes.

Overview

This laboratory explores the USCI and USI communication interfaces in SPI mode. The MSP430 devices included on the Experimenter’s board will exchange messages between themselves, one being the MSP430FG4618 (master) that will control operation of the other MSP430F2013 device (slave). The master, by reading the current state of the slave, will drive the slave to the new desired state, controlling its activity. In this particular case, switching the state of LED3 will be implemented.

Resources

This laboratory uses the USCI module of the MSP430FG4618 device and the USI module included on the MSP430F2013. Both units operate in SPI mode.

The Basic Timer1 of the master device is programmed to switch the status of the slave device once every 2 seconds.

The slave is notified of the arrival of information through the counting end interrupt of the USI module.

The resources used are:

- USCI module;

- USI module;

- Basic Timer1;

- Interrupts;

- I/O ports.

Software application organization

The software architecture for this laboratory is shown in Figure 1.

The master unit is composed of two software modules (Lab2_Comm_1.c):

- The "Main master task" module contains the operation algorithm of master unit;

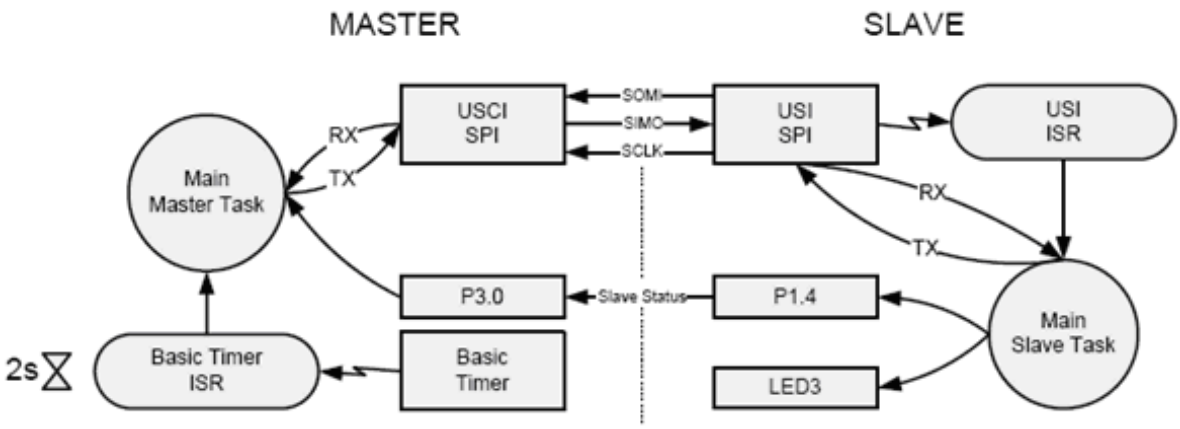
- The "ISR Basic Timer" module wakes the "Main master task" once every 2 seconds.

The slave unit is also composed of two modules (Lab2_Comm_2.c):

- The "Main slave task" module contains the operation algorithm of the slave unit;

- The "USI ISR" module reads the data received, prepares the USI module for new reception and wakes the "Main slave task" to execute the algorithm associated with the reception of the new command.

Software architecture



System configuration

USCI_B (master) control registers configuration

The SPI connection will operate in the following mode:

- Clock phase -> Data value is updated on the first UCLK edge and captured on the following edge;

- Clock polarity -> the inactive state is low;

- MSB first;

- 8-bit data;

- Master mode;

- 3-Pin SPI;

- Source clock -> SMCLK.

The following control registers are configured based on these characteristics:

```
UCB0CTL0 = 0x29;
    //UCB0CTL0 =
    //
UCCKPH|UCCKPL|UCMSB|UC7BIT|UCMST|UCMODEx|UCSYNC|
    //UCCKPH (Clock phase) = 0b -> Data is changed
on the
    // first UCLK edge and captured on the
following edge.
    //UCCKPL (Clock polarity) = 0b -> Inactive
state is low
    //UCMSB (MSB first select) = 1b -> MSB first
    //UC7BIT (Character length) = 0b -> 8-bit data
    //UCMST (Master mode) = 1b -> Master mode
    //UCMODEx (USCI mode) = 00b -> 3-Pin SPI
    //UCSYNC (Synch. mode enable) = 1b ->
Synchronous mode

UCB0CTL1 = 0x81;
    //UCB0CTL1 =
    // UCSSELx | Unused |UCSWRST|
    //UCSSELx (USCI clock source select)= 10b ->
SMCLK
    //UCSWRST (Software reset) = 1b -> normally
set by a PUC
```

Data rate USCI_B (master)

The system clock is configured to operate with a frequency of ~ 1048 kHz from the DCO. This frequency will be the working base frequency of the USCI module. The connection operates at a clock frequency of ~ 500 kHz. Configure the following registers:

```
UCB0BR0 = 0x02;
UCB0BR1 = 0x00;
    // DATA RATE
    // Data rate = SMCLK/2 ~= 500kHz
    // UCB0BR1 = 0x00 & UCB0BR0 = 0x02
```

Port configuration USCI_B (master)

In order to set the external interfaces at the USCI module, it is necessary to configure the I/O ports. Select the USCI peripheral in SPI mode, matching the connections provided at the Experimenter's board:

```
P3SEL |= 0x0E; // P3.3, P3.2, P3.1 option select
```

USI (slave) control registers configuration

The SPI connection will operate on the following mode:

- MSB first;

- 8-bit data.

- Slave mode;

- Clock phase -> Data is changed on the first SCLK edge and captured on the following edge;

- USI counter interrupt enable.

The following control registers are configured based on these characteristics:

```
USICTL0 = 0xE3;
    //USICTL0 =

//USIPE7|USIPE6|USIPE5|USILSB|USIMST|USIGE|USIOE|USISWRST
    //USIPE7 (USI SDI/SDA port enable) = 1b -> USI enabled
    //USIPE6 (USI SDO/SCL port enable) = 1b -> USI enabled
    //USIPE5 (USI SCLK port enable) = 1b -> USI enabled
    //USILSB (LSB first) = 0b -> MSB first
    //USIMST (Master) = 0b -> Slave mode
    //USIGE (Output latch control) = 0b -> Output latch
    // enable depends on shift clock
    //USIOE (Serial data output enable) = 1b-> Output enable
    //USISWRST (USI software reset) = 1b -> Software reset

USICTL1 = 0x10;
    //USICTL1 =

//USICKPH|USII2C|USISTTIE|USIIE|USIAL|USISTP|USISTTIFG|USIIFG
    //USICKPH (Clock phase select) = 0b -> Data is changed
    // on the first SCLK edge and captured on the following edge
    //USII2C (I2C mode enable) = 0b -> I2C mode disabled
    //USISTTIE (START condition interrupt) = 0b -> Not used
```

```
    //USIIE (USI counter) = 1b -> Interrupt
enabled
    //USIAL (Arbitration lost) = 0b -> Not used
    //USISTP (STOP condition received) = 0b -> Not
used
    //USISTTIFG (START condition int. flag) = 0b -
> Not used
    //USIIFG (USI counter int. flag) = 0b -> No
int. pending
```

Analysis of operation

Once the USCI module is configured in accordance with the previous steps, initiate the experiment with the files Lab2_Comm_1.c (master – MSP430FG4618) and Lab2_Comm_2.c (slave – MSP430F2013), compiling them and running them on the Experimenter's board.

For this laboratory, it is necessary to set the following jumper settings:

- PWR1/2, BATT, LCL1/2, JP2;

- SPI: H1- 1&2, 3&4, 5&6, 7&8.

Once the program code is running in the two microcontrollers, monitor LED3 on the Experimenter's board. It will blink with a period of 4 seconds.

This example and many others are available on the MSP430 Teaching ROM.

Request this ROM, and our other Teaching Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp

Laboratory Communications: Lab3 - Echo test using I2C
Using the MSP-EXP430FG4618 Development Tool establish a connection between two MSP430's via the I2C bus. The master receives one byte from the slave. This is the master code. It receives a single byte as soon as a button connected on P1.0 was pressed.

Laboratory Communications: Lab3 - Echo test using I2C

Introduction

The MSP430 contains built-in features for both parallel and serial data communication. This chapter describes the operation of these peripherals, and discusses the protocols, data formats and specific techniques for each type of data communication.

The communication modules available for the MSP430 family of microcontrollers are USART (Universal Synchronous/Asynchronous Receiver/Transmitter), USCI (Universal Serial Communication Interface) and USI (Universal Serial Interface). These provide asynchronous data transmission between the MSP430 and other peripheral devices when configured in UART mode. They also support data transmission synchronized to a clock signal through a serial I/O port in Serial Peripheral Interface (SPI) and Inter Integrated Circuit (I2C) modes.

Overview

This laboratory explores the USCI and USI communication interfaces in I^2C mode. It uses the two MSP430 devices included on the Experimenter’s board: MSP430FG4618 as the master and the MSP430F2013 as the slave. The master receives a single byte from the slave as soon as a button connected to P1.0 is pressed.

Resources

This laboratory uses the USCI module of the MSP430FG4618 device and the USI module included in the MSP430F2013. Both units operate in I2C mode.

The interrupts on the slave unit are generated exclusively by the USI module. They are:

- START condition in the I2C bus;

- Data reception and transmission.

The interrupts on the master unit are provided by the USCI module. They are:

- Data reception;

- Interrupt on Port1.

The resources used are:

- USCI module;

- USI module;

- Interrupts;

- I/O ports.

Software application organization

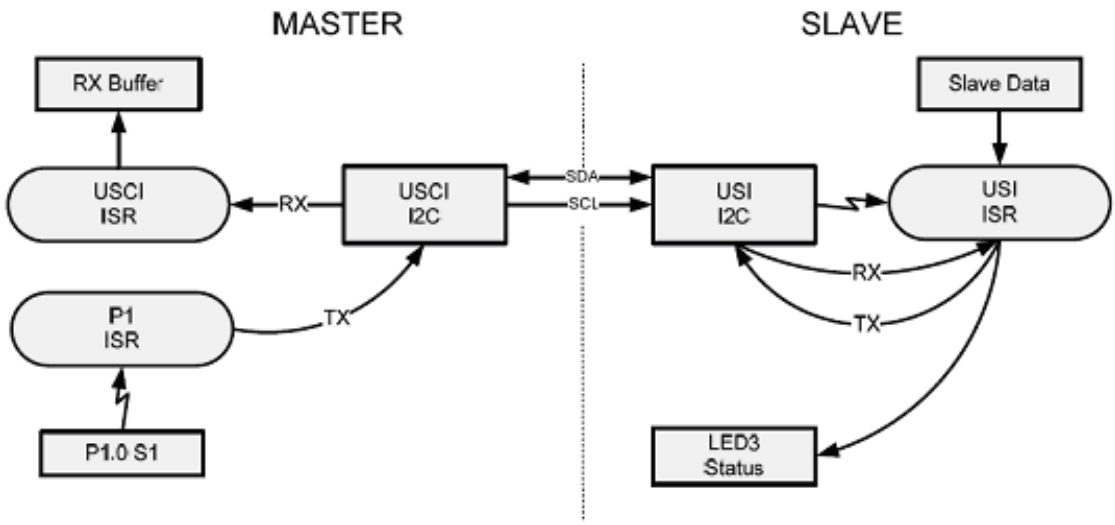
The software architecture for this laboratory is shown in Figure 1.

The master task is composed of two interrupt service routines (Lab3_Comm_1.c):

- S1 switch service routine used to receive a new frame from the slave;

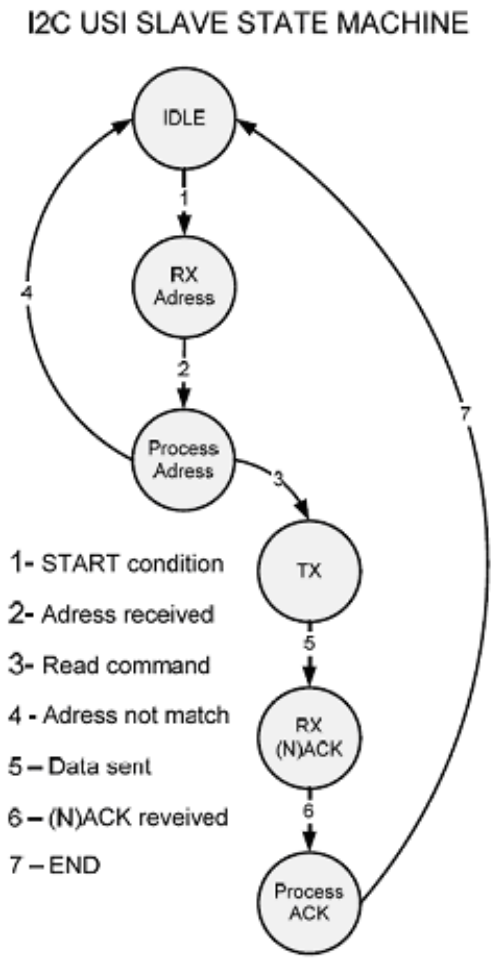
- USCI module interrupt service routine that reads the data sent by the slave.

Software architecture



For the operational capability of the slave unit based on the USI module, it is necessary to implement a state machine as shown in Figure 2. It is important to note that the states “RX Address” and “RX (N)ACK" are transient states that ensure the USI module is prepared for the next activity.

Slave state machine.



System configuration

USCI_B (master) control registers configuration

The connection via I^2C bus will operate in the following mode:

- Address slave with 7-bit address;

- Master mode;

- Single master;

- USCI clock source is SMCLK;

The following control registers are configured based on these characteristics:

```
UCB0CTL0 = 0x0F;
    //UCB0CTL0 = UCA10 | UCSLA10 | UCMM | Unused |
UCMST | UCMODEx | UCSYNC
    //UCA10 (Own address) = 0b -> Own address (7-
bit)
    //UCSLA10 (Slave address) = 0b -> 7-bit slave
address
    //UCMM (Multi-master) = 0b -> Single master
    //Unused
    //UCMST (Master mode) = 1b -> Master mode
    //UCMODEx (USCI mode) = 11b -> I2C Mode
    //UCSYNC (Synchronous mode enable) = 1b ->
Synchronous


UCB0CTL1 = 0x81;

    //UCB0CTL1 = UCSSELx | Unused | UCTR |
UCTXNACK | UCTXSTP | UCTXSTT | UCSWRST
    //UCSSELx (USCI clock source select) = 10b ->
SMCLK
    //Unused
    //UCTR (Transmitter/Receiver) = 0b -> Receiver
    //UCTXNACK (Transmit a NACK) = 0b -> ACK
normally
    //UCTXSTP (Transmit STOP condition) = 0b -> No
STOP
    //UCTXSTT (Transmit START condition) = 0b ->
No START
    //UCSWRST (Software reset) = 1b -> Enabled
```

Data rate USCI_B (master)

The system clock is configured to operate with a frequency of ~ 1048 kHz from the DCO. This frequency will be the working base frequency of the USCI module. The connection operates at a clock frequency of ~ 95.3 kHz:

```
// DATA RATE
// data rate -> fSCL = SMCLK/11 = 95.3kHz
    UCB0BR0 = 0x0B; // fSCL = SMCLK/11 = 95.3kHz
    UCB0BR1 = 0x00;
```

Port configuration USCI_B (master)

In order to set the external interfaces at the USCI module, it is necessary to configure the I/O ports. Select the USCI peripheral in I^2C mode matching the connections provided at the Experimenter's board:

```
P3SEL |=0x06; // Assign I2C pins to USCI_B0
```

USI (slave) control registers configuration

The connection via I^2C bus will operate in the following mode:

- Slave mode;

- USI counter interrupt enable (RX and TX);

- START condition interrupt-enable;

- USIIFG is not cleared automatically.

The following control registers are configured based on these characteristics:

```
USICTL0 = 0XC1;

    //USICTL0 = USIPE7 | USIPE6 | USIPE5 | USILSB
```

```
| USIMST | USIGE | USIOE | USISWRST
    //USIPE7 (USI SDI/SDA port enable) = 1b -> USI
enabled
    //USIPE6 (USI SDO/SCL port enable) = 1b -> USI
enabled
    //USIPE5 (USI SCLK port enable) = 0b -> SCLK
disable
    //USILSB (LSB first) = 0b -> MSB first
    //USIMST (Master) = 0b -> Slave mode
    //USIGE (Output latch control) = 0b -> Output
latch
    // enable depends on shift clock
    //USIOE (Serial data output enable) = 0b ->
Output enable
    //USISWRST (USI software reset) = 1b ->
Software reset

USICTL1 = 0x70;

    //USICTL1 = USICKPH | USII2C | USISTTIE |
USIIE | USIAL | USISTP | USISTTIFG | USIIFG
    //USICKPH (Clock phase select) = 0b -> Data is
changed
    // on the first SCLK edge and captured on the
following edge.
    //USII2C (I2C mode enable) = 1b -> I2C mode
enabled
    //USISTTIE = 1b -> Interrupt on START
condition enabled
    //USIIE = 1b -> USI counter interrupt enable
    //USIAL (Arbitration lost) = 0b -> Not used
    //USISTP (STOP condition received) = 0b -> Not
used
    //USISTTIFG (START condition int. flag) = 0b -
> Not used
    //USIIFG (USI counter int. flag) = 0b -> No
```

```
int. pending
```

The slave unit interrupt service routine is not complete. The portion related to the “I2C_TX” state needs to be completed:

- Configure the USI module as output;

- Insert the information to transmit using the transmission register;

- Configure the bit counter.

```
// USI Bit Counter Register
USICNT |= 0x20;

    //USICNT = USISCLREL | USI16B | USIIFGCC |
USICNTx
    //USISCLREL (SCL release) = 0b -> SCL line is
held low
    // if USIIFG is set
    //USI16B (16-bit shift register enable) = 0b -
> 8-bit
    // shift register mode
    //USIIFGCC (USI int. flag clear control) = 1b
-> USIIFG
    // is not cleared automatically
    //USICNTx (USI bit count) = 00000b -> (not
relevant)

// I2C state machine:
USICTL0 |= USIOE;   // SDA = output
USISRL = SlaveData; // Send data byte
USICNT |= 0x08;     // Bit counter = 8, TX data
```

Analysis of operation

Once the USCI module is configured in accordance with the previous steps, initiate the experiment with the files (Lab3_Comm_1.c) (master – MSP430FG4618) and (Lab3_Comm_2.c) (slave – MSP430F2013), compiling them and running them on the Experimenter's board.

For this laboratory, the following jumper settings are required:

- PWR1/2, BATT, LCL1/2, JP2;

- SPI: H1- 1&2, 3&4.

The slave data is sent and increments from 0x00 with each transmitted byte, which is verified by the Master. The LED is off for address or data Acknowledge and the LED turns on for address or data Not Acknowledge. LED3 blinks at each data request. It is turned on with a START condition and it is turned off by the data transmit acknowledge by the slave (Note: the I^2C bus is not released by the master since the successive START conditions are interpreted as "repeated START").

Verify the value received setting a breakpoint in the line of code "`RxBuffer = UCB0RXBUF;`" of the USCI interrupt.

This example and many others are available on the MSP430 Teaching ROM.

Request this ROM, and our other Teaching Materials here https://www-a.ti.com/apps/dspuniv/teaching_rom_request.asp